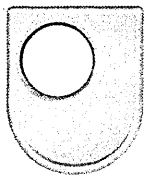
Analysis and Development of Instrument Software Paradigms: Conception and Implementation of a New Instrument Control and Data Acquisition System, Proven by Material Scientific Applications

Thesis

UNRESTRICTED



PhD Thesis

Analysis and Development of Instrument Software Paradigms; Conception and Implementation of a new Instrument Control and Data Acquisition System, Proven by Material Scientific Applications.

Stefan Alexander Flemming

Open University Milton Keynes

A thesis submitted to the Department of Maths, Computing and Technology of the Open University for the Degree of Doctor of Philosophy

2013-06-03

Date of Submission: 30 September 2012
Date of Award: 4 July 2013

ProQuest LLC.
789 East Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106 – 1346

Acknowledgements

I would like to express my gratitude to my doctor father *Lyndon Edwards* and to my supervisor *Jon James*, whose expertise, patience and helpful tips added considerably to my graduate experience. I appreciate the time at the Open University Milton Keynes, conferences and the manifold video meetings where *Jon James* was always willing to give me feedback, answered my questions and gave me valuable intellectual approaches. These thanks also extend to the head of the department *Mike Fitzpatrick*, my third-party monitor *Mark Endean* and the staff of the department *Materials Engineering.*

Very special thanks go to my local colleagues at the HZB and the staff at the FRM-II who made it possible to use the instruments and gave me support at any time. My acknowledgements especially go to my external supervisor *Rainer Schneider*, my colleagues *Robert Wimpory*, *Jens-Uwe Hoffmann*, *Florian Henkel*, *Mirko Boin*, *Tobias Poeste* and all other staff members. This also includes *Lutz Rossa* and *Olaf P. Sauer* from the HZB IT team. For their confidence in using Open Inspire at their instruments I thank *Michael Hofmann* (FRM-II), *Andrew Venter*, *Morney van der Watt* (NECSA *SA*) and *Ulf Garbe* (ANSTO *AU*).

Several students helped me to test Open Inspire, find bugs or used the functionality of OI for their diploma theses or seminar papers. I especially thank *Martin Schobert* for his review and help in connection with the OI *Calibration System* and *Network Service*, *Fabian Holstein* for the maintenance of the OI servers, *Hakan Yavuz* for his engagement in finding bugs in connection with the NeXus implementation, *Markus Lorenz* for his optimizations of the *Levenberg Marquard Fits*, *Christian Randau* for his texture analysis modules *(diploma thesis)* and *Thomas Weiß* for his help as graphic designer.

My very best acknowledgments however go to my parents and my friends. Without the support and patience of my mother *Sabine Flemming* and my father *Michael Flemming* I would never have been able to write the thesis! Thank you for your help at any time, your encouragement and your financial support. Thank you *Robert Westenkirchner* for the proofreading and all my friends for your incredible patience.

Abstract

During the last 50 years, the quality of analysis methods in many scientific disciplines has been enhanced by electronic applications, automation and data processing. While the features, performance and usability of these processes have been continually enhanced, it is conspicuous that the majority of institutes operate own proprietary software. This situation arises for both historical and financial reasons, plus a wish to retain autonomy fuelled by the requirement for a system that remains compatible with both new and legacy hardware.

This thesis reviews the commonly used scientific software systems and their stakeholders and tries to identify generic problems. The demands on instrument systems are summarized by a requirement specification. Based on these requirements, a basic concept is developed that reflects the current state-of-the art in software design and which may provide a blueprint for instrument system architectures.

The results are used to create a proof-of-concept implementation. Core to this approach is an application server that comes with a container, which makes use of the Inversion-of-Control pattern to loosely couple and execute components. These do not need to implement fixed interfaces and are thus decoupled from a specific use-case. Components can, for example, be proxies that control and acquire data from legacy hardware, perform calculations, provide a human-machine interface or act as storage. They are dynamically wired to experiments using *XML*-based Assembly files. Both *Assemblies* and *Components* can be published using a central store on a collaboration platform and shared by the community. This increases reusability and allows the use of existing Assemblies with new hardware by simply replacing the hardware proxy modules.

Example components have been provided for the access to legacy and new instrument hardware, the storage of results in the *NeXus* format, data reduction, simulation with *McStas*, the execution of customizable scans and the visualization of data.

CONTENTS

CHAPTER

ONE

PREFACE

1.1. Survey of the Thesis

CHAPTER 1 - PREFACE - [PAGE 11 FF.]

The first chapter starts with introductory notes and suggestions how to read the thesis. Information is given about the typographical conventions, how to navigate through the thesis and how to obtain the developed software Open Inspire.

CHAPTER 2 - REVIEW OF EXISTING RESEARCH AND THEORY- [PAGE 17 FF.]

Chapter two begins with a general description of the research question, a basic description of the experiment physics and an overview to the utilized research environment. This introduction is followed by a comprehensive review of software systems used for instrument control, data analysis, experiment simulation and data storage.

CHAPTER 3 - SOFTWARE REQUIREMENTS - [PAGE 55 FF.]

The third chapter covers the requirements for a state-of-the art instrument software. It summarizes the analysis process and goals of the project, identifies and compares the stakeholders, covers the project constraints, relevant facts, scope and context of the work.

The chapter closes with a short summary of the *Functional Requirements* while a comprehensive list of all *Non-Functional Requirements* can be found in Appendix 2 on page 308.

CHAPTER 4 - OPEN INSPIRE - THE IMPLEMENTATION - [PAGE 71 FF.]

Chapter four covers the concrete realisation of a software architecture in consideration of the stated requirements of chapter three. It starts with the basic concepts of the system, which includes network topology decisions, the introduction of the application server architecture and important implementation details. This introduction is followed by the design of components, their management and their wiring to so-called *Assemblies* as well as the summary of global system services and configurations. To make components and documents publicly available, a collaboration platform has been created, which is shortly covered before the chapter finally closes with a summarization of the build system and IDEs integration.

CHAPTER 5 - OPEN INSPIRE IN THE FIELD - [PAGE 175 FF.]

The fifth chapter proves the concept by presenting component implementations that cover all important parts of an instrument system. It starts with an introduction of User Interfaces, storage of experiment data using the *NeXus*-format, customizable scan implementations, control of legacy hardware, simulation of instruments using McStas and closes with an automated instrument calibration setup.

CHAPTER 6 - SUMMARY, CONCLUSION AND OUTLOOK - [PAGE 255 FF.]

Chapter six summarizes the results and considers the extent to which the software Open Inspire fulfils the requirements of the research community. The assets are evaluated as well as the limitations and an outlook is given how the developed software can be enhanced.

APPENDIX - [PAGE 268 FF.]

For the sake of readability, continuative information can be found in the appendix. It is the location for configuration files, relevant source code, the *non-functional* requirements and additional graphics.

Finally the thesis closes with the bibliography, list of figures, list of tables and a glossary.

1.2. Readers Instructions

This section contains useful information that simplifies navigating through the document and understanding of some typographical conventions.

Navigating through the Document

Some techniques have been utilized to provide an improved navigation between parts of the thesis. These simplifications affect the electronic document as well as the hardcopy.

LITERATURE REFERENCES

A literature reference is marked by a label in squared brackets. The corresponding entry can be found in the bibliography in the appendix at the end of the thesis. To make it easier to find a cited document, a URL that contains the latest web location of the cited document is given whenever possible[1]. If the document is read at the computer, clicking cite marks and web links is supported to directly navigate to corresponding bibliography entries and related documents. Here is an example for a citation of the Open Inspire *NOBUGS* paper →[FJS+08]←. To find the text passages where a document is cited, back links are provided.

ACRONYMS AND GLOSSARY

In IT related projects it is common to use acronyms for technologies, program names or software paradigms. In the majority of cases the first occurrence of an acronym is written in its completed form followed by its abbreviation in brackets, e.g. →Information Technology (IT)←. Later on the acronym is only written in its short form IT. All acronyms are added to an index of acronyms at the end of the document to consult the meaning even when its written in short form. Similar to the bibliography, backlinks are provided to find the passages in the text where the acronym is used. The electronic document allows clicking the acronym and navigating to the explanation as well as clicking the indexed pages and showing the occurrence.

[1]Web links have the drawback that the referenced document might have changed or is no more available. It is thus recommended to use a service such as the *WayBackMachine* (`http://archive.org/web/web.php`) to obtain a cached version of the document at the effective date of the cite.

SECTION AND EXTERNAL REFERENCES

This thesis makes use of references to other sections or external web resources. An example of such a reference is →chapter 6 on page 255← or →http://www.openinspire.org←. It is possible to quickly navigate to the related text position by clicking it.

REQUIREMENT REFERENCES

Appendix 2 (page 308 ff.) contains a list of *Non-Functional* software requirements that are referenced in the running text. The printed text allows the reader to find the requirements with help of their unique id. If an electronic reader is used, it is possible to jump to a requirement by clicking the related link. Such a link is printed in the form: [☒ 3.2.2].

Understanding the Diagrams

Wherever applicable, meaningful illustrations are used in favor of textual descriptions or source code. The first choice for software-related diagrams is the Unified Modeling Language (UML), which allows modeling *Object Oriented* software and its environment in a standardized way. To provide a better readability, the illustrations make use of the liberty to mix different types of diagrams and to hide irrelevant information. Several books have been used as reference for the creation of the included UML diagrams[2] but for the understanding of the used diagrams it is sufficient to read the book *UML 2.0 in a Nutshell* [PP05].

1.3. Software Installation

Within the scope of the thesis, an instrument software has been developed according to the requirements, identified in chapter 3. The software is called Open Inspire[3] and abbreviated in this document as OI. It is published as a freely available *Open Source* product [☒ 7.1.2] and can be downloaded at OI's community portal `http://www.openinspire.org`. This page also contains documents with up-to-date information about installation, configuration and usage of Open Inspire. Beyond that, collaboration tools are available that allow to browse sources, communicate with developers, report bugs and get access to documentation.

[2]UML books used for this thesis:[Bal01, Bal05, BRJ06, ER02, ER05, Sch03, uml08]
[3]Acronym for Open Infrastructure for Natural Science and Programming in Research Environment.

1.4. Internal Section

The OI webpage comes with restricted area, which provides supplemental information for this thesis. If a reader prefers to read the thesis on a tablet or computer, it is possible to download the document in form of a PDF file. The URL to the internal section is `http://thesis.openinspire.org`. This section also contains links to additional information or demonstrations. It is recommended to regularly check the page for updates or demo material.

The page is protected and can be accessed using the username **`thesis`** and password **`phdthesis`**.

CHAPTER

TWO

REVIEW OF EXISTING RESEARCH AND THEORY

The intention of the second chapter is to familiarize the reader with the research topic, the research environment, the state-of-the-art in experiment related software, and the identification of demands for new software functionality.

The chapter starts with the definition of the general research question that will be refined using a comprehensive requirement analysis in chapter 3.

Subsequently a short overview presents the basic physics and experiments performed in the selected research and test environment.

This overview will be completed by a short introduction to the assembly of the instrument and sample environment.

The next section covers commonly used experiment related software and gives a comprehensive survey of instrument control and data acquisition software, data analysis, processing and visualization software, instrument and experiment simulation as well as software for data exchange and storage.

Finally this chapter closes with an analysis of software demands achieved by user and developer interviews.

2.1. The Research Question

Using neutron and synchrotron radiation for non-destructive material analyses in the area of material science and engineering is a common technique. In addition to X-ray experiments, synchrotrons and neutrons can be used to get a deep insight into materials without manipulating their geometry or characteristics.

All of these experiments have a similar setup and use diffraction techniques for the investigation of material properties. The simplest setup consists of a source, a sample and a detector. The source provides a synchrotron or neutron beam that will hit the sample, which will scatter, absorb or transmit the beam. Intensities, positions and distributions of the scattered radiation, measured by the detector are used to infer information about constitutional and structural properties of the material.

Generating synchrotron or neutron radiation by accelerators, reactors and spallation sources involves high complexity and costs, which makes beam-time rare and expensive. An important target is hence to reduce the duration of experiments by optimizing the setup and by decreasing idle-times. While one way to achieve this is by hardware upgrades, such as enhanced neutron optics or more sensitive detectors, this thesis seeks to answer the question as to the extent this can be achieved by means of enhanced software systems.

Software can speed-up experiments and increase quality by automating recurring tasks, minimizing breaks necessary for user-input as well as utilizing algorithms that decrease the duration of experiments by running them autonomously. For the overall benefit it is however necessary to also include the development effort for such new software functionality into the consideration if such an upgrade is worthwhile.

Since scientific experiments are often specialized for a particular purpose, continually enhanced and tailored to a specific environment, they are not developed for a mass-market. This means that nearly all institutes invest time and money in proprietary in-house solutions. The result are similar redundant functionalities that are over and over developed at different operating sites.

This redundancy is not necessary but difficult to prevent since there is no generally accepted standard, which is compatible with all possible instrument setups. And if such a standard would exist, it is nearly impossible to raise up the effort in changing all existing setups and upgrading them with environment-specific functionality.

The thesis hence seeks to answer the question in which way a system can be designed that reduces development redundancy by providing means that allow to share common functionality between institutes. The challenge is here to allow a smooth integration of existing concepts and to be compatible with legacy systems with a minimal intervention into existing infrastructures. Beyond that, developing a standard is a community process and cannot be established by an individual, so that the approach must be flexible enough to incorporate upcoming standards without the need to make changes to the new software system itself.

The objective is hence to develop a functional system as well as example implementations in the area of neutron diffraction, which enrich the features and usability of experiments without forcing users to forgo their familiar experiment environment. User Interfaces, storage mechanisms, advanced scans, the integration of legacy systems, the simulation of experiments and the introduction of an automated instrument alignment shall thereby be considered as well as the integration of cloud-based collaboration methodologies.

2.2. Material Scientific Backgrounds

The idea of the structure of materials was a long treasured secret of the nature. More than 2000 years ago, the philosopher Plato (427-347 B.C.) tried to explain the behavior of materials and proposed that everything consists of small composed particles, cubes, tetrahedrons, octahedrons and icosahedrons. In the last hundred years new analysis methods gave us the opportunity to look deeper into the material and shed light on the mechanisms and structures. The new branch of research *Crystallography*[1], originated as the science of the study of crystal forms, covers exactly these concerns, the study of atomic arrangements in crystalline materials.

[1] crystallon (Greek) = cold drop / frozen drop

2.2.1. Basic Crystallography and Diffraction Physics

Today, we know much more than ever about the basic physics that describe the properties of materials. While an amorphous solid is a solid in which there is no long-range order, crystalline materials consist of structures that recur periodically. These structures exist in many types of materials, which may be classified by their crystallographic structure. The arrangement of atoms in such structures can be described as unit cells, spatial arrangements of atoms that are stacked in a three-dimensional space. The length of the cell edges and angles between them defines the lattice parameters of the cell. Fourteen so-called bravais lattices can be used to summarize all existing types of lattices. Many specific characteristics of a material can be inferred from the specific lattice type of a material. [KBB98]

The three common Diffraction Techniques

Three common non-destructive measurement techniques used in materials science and engineering are X-ray-, synchrotron and neutron diffraction [Sun07]. Prior to these techniques, the study of crystals was based on geometric measurements of the crystals using a goniometer. This is a mechanical instrument, which can rotate objects to precise angular positions and measures angles to be plotted in a stereographic projection. Each point is labelled by a so-called Miller index, a triple of integers that denote planes orthogonal to a direction in the basis of the so called reciprocal lattice vectors.

Modern techniques now depend on the analysis of diffraction patterns that can be measured after targeting an X-ray, synchrotron or neutron-beam to the sample. All three types of radiation interact with the specimen in different ways, so the most appropriate technique can be selected for the experiment. While X-rays for example interact with the spatial distribution of valence electrons, neutrons are scattered by the atomic nuclei through strong nuclear forces and additionally by magnetic fields since its magnetic moment is non-zero. Synchrotrons may be used whenever conventional X-ray tubes are not sufficient because of the need for a higher brightness and intensity[2].

[2]The intensity is many orders of magnitudes more than X-rays produced in conventional X-ray tubes

Details of Neutron Diffraction

The first known reference, where neutrons are discussed to study applied problems of this sort is titled *Applications of Neutron Diffraction of Non-Destructive Testing Problems* and published by the former US National Bureau of Standards (NBS)[3] in 1976/77 [PCAE78]. After the Institute Laue-Langevin (ILL) was commissioned in 1971, European and Australian scientists started making routine measurements with neutrons. First depthprobing measurements began 1979/80 in Missouri, Karlsruhe and Harwell. In the following years the community started to grow and neutron scattering became a widely used technique for non-destructive material analysis.

A good starting point for more detailed information to neutron diffraction are the publications *The Early History of Neutron Stress Measurements* [Kra08], *Introduction to diffraction in materials science and engineering* [KA01], *Measurement of residual and applied stress using neutron diffraction* [HK92] and *Neutron stress measurements in the 21st century* [Kra01] .

2.2.2. Neutron Sources for Material Scientific Experiments

The rising interest in neutron experiments caused the commissioning of numerous neutron sources worldwide. The types of sources can be generally classified in two branches, fission reactor and spallation sources. [FL03]

Fission reactors

At present most neutron sources are based on steady state reactors that are optimized to produce a high neutron flux. In most cases, the core consists of enriched 235U isotope that releases neutrons due to the fission process. Neutron beam tubes that are directed towards the reactor core are used for transportation of neutrons to the instrument. A moderator between core and tubes is used to slow the neutrons so they have a suitable wavelength for probing the atomic distances in the materials under investigation. Usually monochromatic instruments are found at fission reactor sources due to the steady state neutron flux.

[3]The NBS is today the National Institute of Standards and Technology (NIST)

For residual stress analyses[4] generally only the position of one interference peak is detected to evaluate the local lattice strain.

Spallation sources

Neutron spallation sources use a cyclotron setup to accelerate high-energy protons. These hit a target with a high atomic mass and cause subsequent nuclear reactions that put the target nuclei in a highly excited state from which it decays by "evaporating" neutrons. The resulting pulsed neutron beam enables the powerful time-of-flight method, which delivers the complete diffraction pattern in a single measurement.

2.2.3. Neutron Diffraction Applications

More and more modern engineering progresses profit from developments of new materials and innovations in their processing and treatments. Materials characterization techniques for the study of metals, alloys, ceramic and composites, especially non-destructive analyses of residual stress profiles and textures, have gained increasing importance. Thus a large amount of beam time at the materials research diffractometers used for the practical part of this thesis is exclusively used for industrial research. Among the components that were investigated are crankshafts, impellers, turbine blades, pistons, cylinder heads, and welds.

Residual Stress Measurements

Residual stress is the stress remaining in a component or structure when there are no external forces acting on it. They may be generated or modified at every stage in the component life cycle, from original material production to final disposal. Welding, for instance, is one of the most significant causes of residual stresses. The measurement and analysis of residual stresses has gained significant importance over the past years due to the increasing demands in improving the properties of new engineering materials and components. The ability to measure these residual stresses accurately will thus lead to the manufacture of stronger, lighter and cheaper components by industry. This drive to optimize material performance whilst minimizing component weight will ensure that this field continues to grow. In future,

[4] An explanation of residual stress measurements follows on page 22

neutron scattering will gain further importance for strain measurements as new dedicated neutron diffractometers become available. These instruments will allow the utilization of the full potential of the technique and are likely to further establish neutron stress analysis as a routine method for industry. Unlike conventional X-rays, neutrons can penetrate deep into most crystalline materials. This makes the neutron a non-destructive probe, capable of measuring in the bulk of a material specimen under investigation. Neutron diffraction can measure stresses in three dimensions with a typical spatial resolution (1-5 mm). No other stress measurement technique offers a combination of these properties [Hut05].

Texture Measurements

A large number of polycrystalline materials, either manmade or natural, display a non-random orientation of crystallites; known as texture. This texture has a profound effect on the material, making physical properties, such as elastic moduli anisotropic [WH04]. There are two types of texture; lattice preferred orientation (LPO) also known as 'preferred crystallographic orientation' and shape preferred orientation or 'preferred morphological orientation'. In this context only the first type is of relevance due to its accessibility by diffraction experiments. Texture can be measured in several ways, depending on the relationship between the size of the crystal (or grains) and the sampling volume and the size of the specimen. One way is to measure the average texture over a large volume. A second way is to measure with smaller volumes, providing information of texture as a function of position. A third way is to measure the orientation of individual crystals. Texture analysis covers a wide range of research interests, such as engineering[5], physics[6], biology[7] and geology[8]. Diffraction techniques are the most widely used means of measuring texture [Bun85]. However normal X-rays are limited to the surface of a material due to its low penetration. Neutrons offer advantages since they allow the integrated or spatially resolved measurement of bulk specimens because of their high penetration. In-situ experiments are also possible using different sample environments such as *stress-rigs*, *furnaces* or *pressure cells.*

[5] e.g. welds, thin films and car body panel manufacture
[6] e.g. high temperature semi conductors
[7] e.g. bones and shells
[8] e.g. seismic anisotropy

Neutron diffraction texture measurements can be performed at reactor sources, typically using wavelengths of approximately 0.1 to 0.2 nm produced typically by a single crystal monochromator. The sample can be placed within an Eulerian cradle or on a robot at the center of the neutron diffractometer and rotated over the entire orientation range.

2.3. The Research Environment

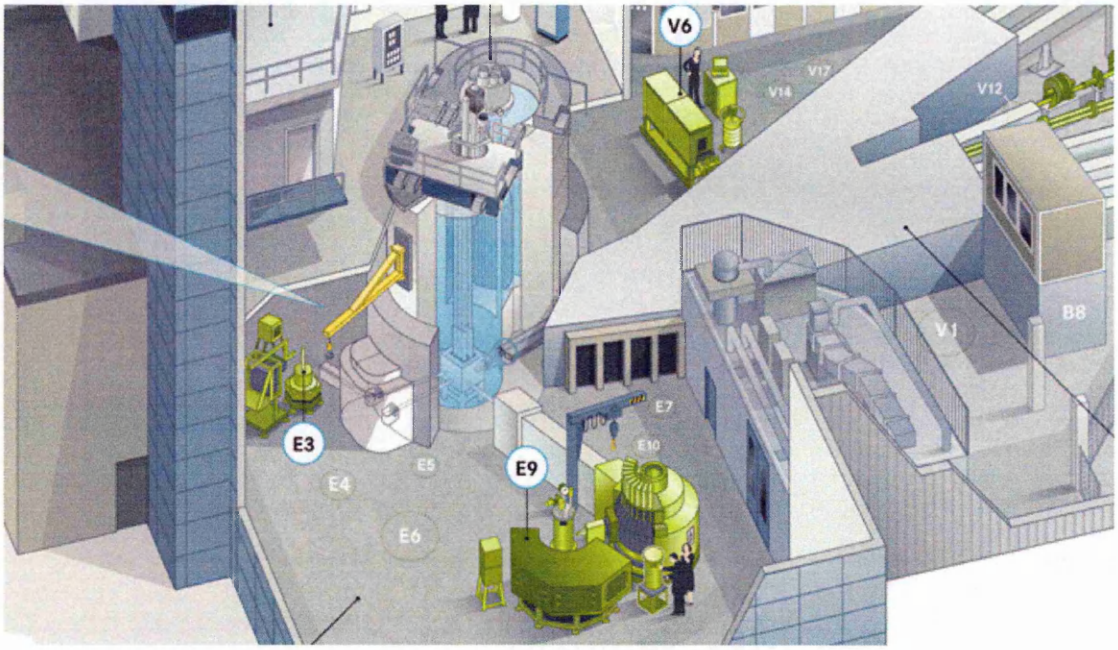


Figure 2.1.: *Experimental Hall at BER-II (From HZB Media Library)*

This PhD-thesis comes with a strong practical part. The development of the software Open Inspire, as described in chapter 4, as well as the practical examples covered in chapter 5, presuppose a capable development environment. To meet this requirement the Helmholtz-Centre Berlin for Materials and Energy (HZB) kindly provided access to the two dedicated residual stress diffractometers *E3* and *E7* located at the research reactor BER-II in Berlin, Germany. Beyond that, the Technical University Munich (TUM) offered intermittent access to the diffractometer *StressSpec*[9] for texture measurements.

[9]located at the Research-Neutronsource Heinz Maier-Leibnitz (FRM-II) in Garching, Germany

2.3.1. The Diffractometer Construction

Both instruments *E3* and *E7* in Berlin as well as *StressSpec* in Garching are nearly identical[10] in their construction. For this reason it is fully sufficient to only describe the general instrument assembly to understand the work in this thesis.

The short overview will describe the way of Neutron starting at the Neutron Source and finally ending at the detector or beam stop of the instrument. By way of example the instrument *E7* has been selected since it has been used for nearly 90 percent of all tests.

The Neutron Source and Beam Tubes

The HZB and TUM both operate fission reactors[11] as neutron source. Figure 2.1 shows the HZB fission reactor BER-II in the oval at the center surrounded by the Experimental Hall (E-Hall). Nine beam-tubes transport the neutrons to the instruments inside the E-Hall. Here the instruments *E3* and *E7* are located. *E3* is illustrated on the left next to the reactor.

The Casemate - Container for the Monochromator and Collimator

Figure 2.2 shows the construction of the instrument. The large gray cylinder on the right is the casemate. Depending on the instrument a number of selectable monochromators and collimators are mounted behind the shielding. Collimators look similar to a comb and are used to filter neutrons with different directions so that all resulting neutrons leave it in parallel. Monochromators consist of crystals that scatter the incoming neutrons in such a way that only neutrons with a particular wavelength are passed.

Immediately before and behind the casemate beam-shutters are mounted. These shutters can interrupt the beam on user demand whenever work at the casemate or instrument need to be performed. Neither shutters are controllable by software.

[10] The few differences are primary caused by different monochromator or collimator setups, space proportions, size of sample tables, height of the incoming beam or the detector resolutions

[11] See section 2.2.2 on page 21 for more details on fission reactors

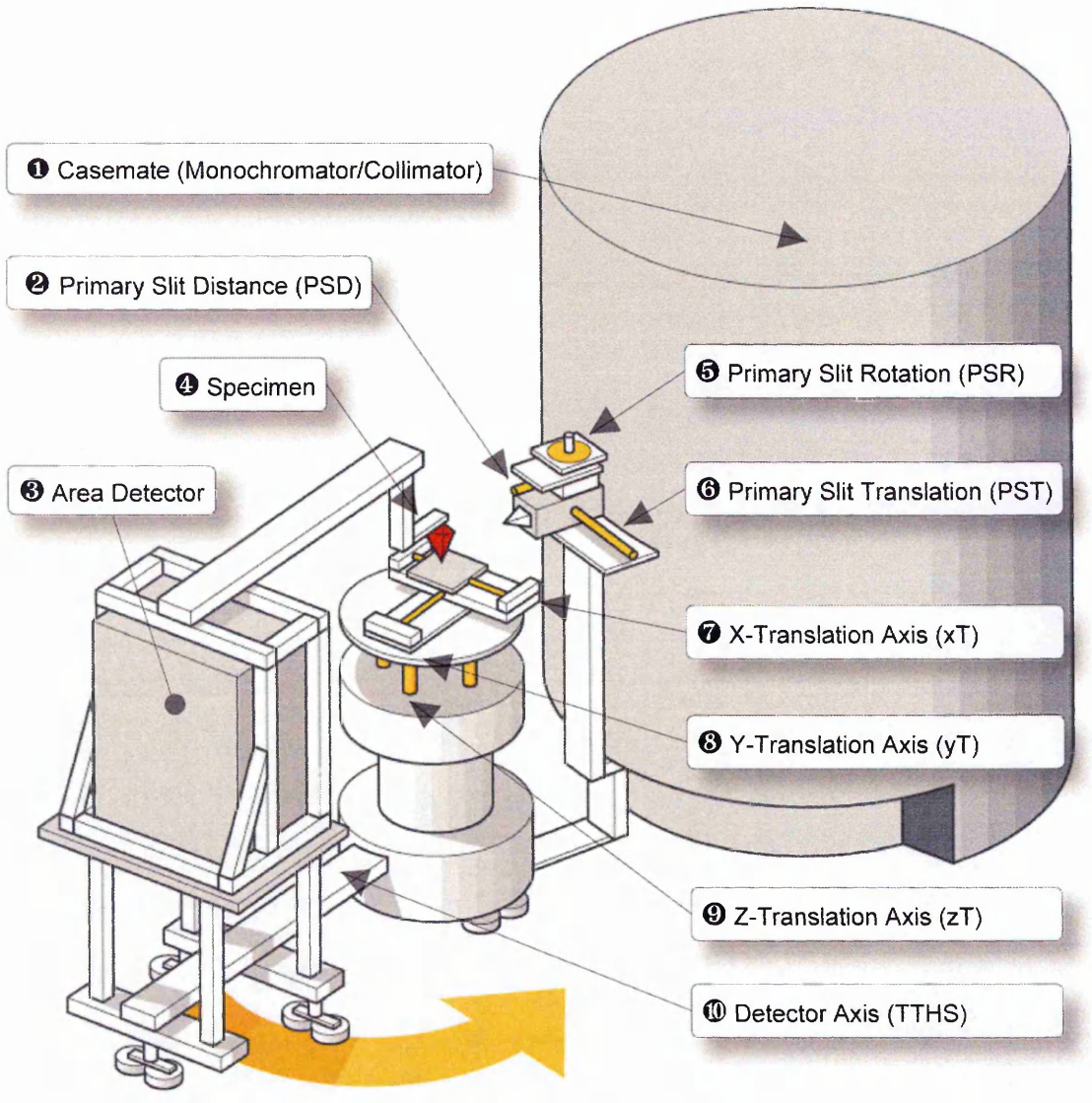


Figure 2.2.: *Neutron Diffractometer (E3, E7, StressSpec)*

The primary slit

Figure 2.3 shows the so called primary slit and the secondary shutter. The beam passes through the opened shutter and will be constricted by the geometry of a mountable slit. To restrict the geometry of the beam diameter three positioners can be used to freely place the slit inside the way of the beam. The slit is made of a material that absorbs all neutrons that are not passed through the hole and so allows a precise portion of the beam to be selected. The first axis is called Primary Slit Translation (PST) and used to move the slit in parallel to the secondary shutter. The second axis is Primary Slit Rotation (PSR) and can rotate the slit to select neutrons with a specific direction. Finally the axis Primary Slit Distance (PSD) moves the slit in parallel to the shutter.

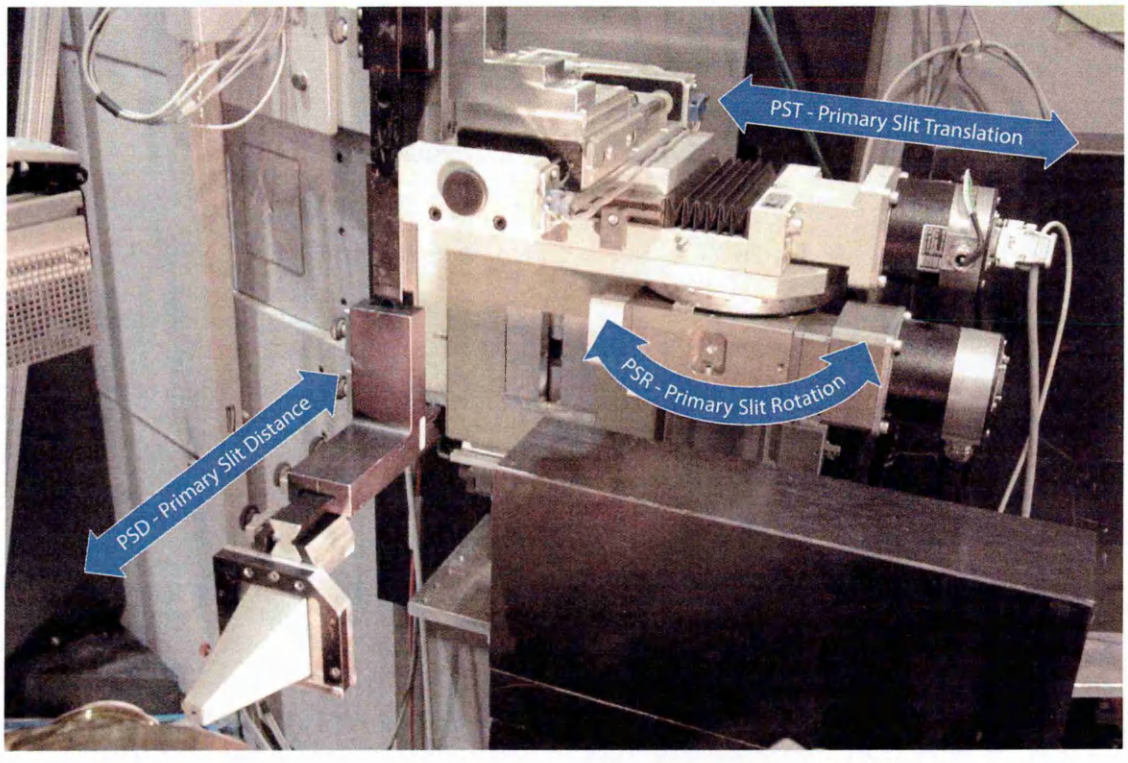


Figure 2.3.: *Primary Slit and positioner axes for translation and rotation*

The Sample Positioning System

The next important component inside the beam is the specimen that is to be investigated. Figure 2.2 shows the specimen in form of a red crystal. Due to the manifold geometries and weights, the investigated sample is mounted on a table that allows to measure even weighty samples with several hundred kilos. Furthermore it is equipped by four motors and related absolute encoders that allow to freely position the sample and read back the positioning values. Figure 2.4 and 2.5 show more detailed views of the sample table. The axes xT and yT translate the sample in x and y direction. zT lifts the table up and down and omgs is used for the rotation.

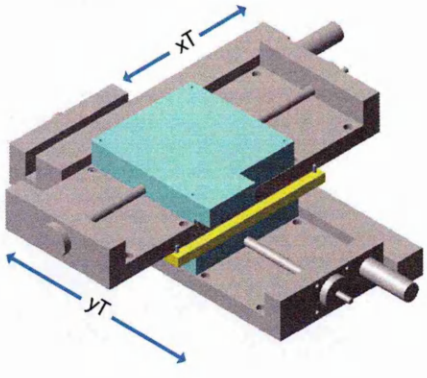


Figure 2.4.: *Translation Axes - xT, yT*

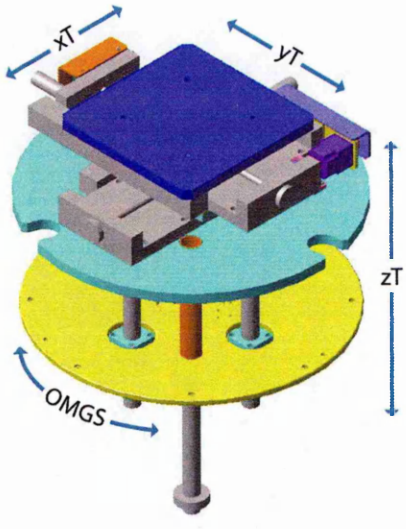


Figure 2.5.: *Goniometer Table*

When these given axes are not sufficient, an *Eulerian Cradle* can be mounted that additionally provides axes to rotate the sample around *PHI* and *CHI*. Figure 2.7 shows an *Eulerian Cradle*, which is additionally equipped with a small sample table and the translation axes xE and yE.

With help of the primary slit and all axes of the sample table it is possible to precisely select the part of the sample that is to be measured and to bring it in the correct measurement position and angle.

The Detection System

The illustrated instrument is equipped with three types of detectors. The first detector MON is directly located behind the secondary shutter and counts the incoming neutrons.

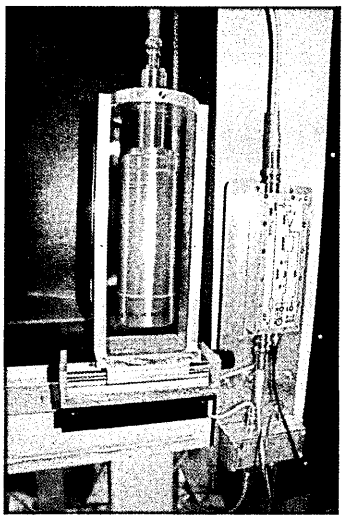


Figure 2.6.: *Singledetector*

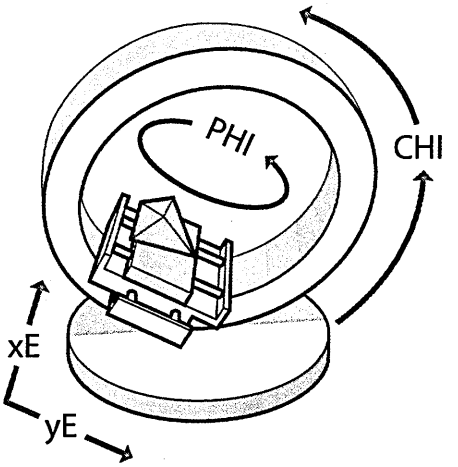


Figure 2.7.: *Eulerian Cradle*

The second detector ADET is an area detector with 128x128 channels[12] and illustrated as grey box on the lower left side of figure 2.2. It detects neutrons scattered by the sample and is for that purpose circular movable around the sample table. The name of this motor driven rotation axis is ttths.

The last detector is the Single Detector (SDET), a plain counter tube for neutron detection. It is placed directly inside the beam behind the sample and used to detect all neutrons that are not diffracted. Its primary use are calibration measurements used to locate the position and direction of the beam. Figure 2.6 shows the detector of instrument *E3* directly in front of the beam-stop. The beam-stop is always the last instrument component in the way of the beam and used to shield the environment from hazardous radiation.

Camera System and Lighting

The final important part of the construction setup is the camera and lighting system. This is the most recently installed system and has been mounted specifically for instrument cali-

[12] *E3* utilizes a detector with 256x256 channels

brations performed by Open Inspire. The setup is composed of a fixed focus camera[13] that delivers a monochrome picture with a resolution of 1280x1024 pixels and a diffuse lighting. For the calibration process a sample with a fixed geometry is mounted on the goniometer, illuminated by the diffuse lighting which casts a shadow of the object on the camera. This camera can now detect the white to black gradients to determine the location of the object on the goniometer. More detailed information to the setup and calibration process can be found in chapter 5.6 on page 244.

2.3.2. The Control Electronics

The link between the mechanical components and the instrument software is the electronics. Although all three instruments *E3*, *E7* and *StressSpec* are nearly clones in their construction, they vary in their electronic setup. The primary difference here is the bus system used for the data transport. *E3* uses the CAMAC-bus[14][cam] while *E7* and *StressSpec* are equipped with a VME-bus[15][vitb]. Nearly all bus components such as the motor-controls have been developed in-house at the HZB and are hence proprietary devices.

Positioning Electronics

All motor axes relevant in this thesis are controlled by the HZB development *ST169*. A microstep power amplifier and linear controller for stepper motors that support software driven and manual motor control. Further features are micro step accurate positioning, operation of end-switches as well as engine idle and short circuit-proofing by means of voltage and current tracing.

Beyond that the axes xT, yT, zT, omgs and tths have an encoder to read back the position value (even after complete instrument shutdowns) thereby making reference-drives obsolete. Internally the proprietary components *SV3901* for *VME* and its *CAMAC* equivalent *Y66* are used as comparators between the position setpoint read from the *EKF-module* and the value read from the encoder. The *EKF-module* is a multiplexer and address-converter be-

[13]Camera Type: M4+CL of company Jai

[14]Computer Automated Measurement And Control (CAMAC) is an international standard for a modular data handling system, developed for data acquisition and experiment control in nuclear physics

[15]The Versa Module Eurocard (VME)-bus is a standardized computer bus developed in 1981. It is physically based on Eurocard sizes, mechanicals and connectors and uses its own signalling system.

tween hardware and VME-station at *E7* and *StressSpec*. For the *E3* CAMAC system a so called *Parallel-Bus-Crate-Controller* combines the functionality of the *EKF-module* and VME-station equivalent.

A motor positioning can so be considered as comprising the following steps; the presetting of a motor setpoint, the activation of the motor controller which drives the motor, the comparison of the setpoint and the actual value and the stopping of the motor when the value is reached.

Detector Electronics

The detector electronics is mounted inside a rack on the reverse of the detector. It contains a high-voltage generator with 3750V as well as several delay and converter circuits. Depending on the instrument the data is directly made available to the VME system or published by an additional server, which is equipped with a PCI detector interface.

Camera Electronics

For the handling of the data detected by the camera, a Windows based camera server is used. Here a PCI frame grabber[16] is used to record the digital camera signals and makes them available to the Operating System for further processing. More information on the calibration process can be found in chapter 5.6.

A more detailed description of the positioning, detection and camera system can be found in chapter 4 of the diploma thesis [Fle05].

2.3.3. The Control Software

The final component in the control chain is the instrument control software. All three of the evaluated instruments and nearly all other instruments at the HZB are controlled by the same software. This software is named *Caress* and provides a unique user interface, several scan algorithms and a large number of hardware drivers to address all HZB instruments in the same way.

[16]The used frame grabber is a *Siso Microenable III* from Stemmer Imaging

This section does not cover *Caress* since section 2.4 on page 32 already summarizes the functionality and chapter 5.4 on page 206 explicitly explains the control of *Caress*-devices with Open Inspire.

2.4. Software Systems

During the last 50 years, the quality of measurement and analysis methods has been enhanced by means of modern electronic applications, automation and data processing. These enhancements enable the collection of more information about the measured material than ever before. Over the past 30 years various software systems for instrument control and data analysis have been developed by different institutes all over the world. The range of these systems goes from simple monolithic command line tools to high level distributed enterprise applications in heterogeneous networks. To illuminate the differences in software design that various instrument environments, requirements and programming skills afforded, it is meaningful to give an overview about popular existing systems. In contemporary software design a couple of paradigms are mandatory. One of these paradigms is the reusability, extensibility and exchangeability of software parts and is directly linked with the creation of a modular, component based system. Further the system has to be failsafe, well maintainable and user friendly. Theses aims are not new; the difference is in the method of their realization. The following paragraphs show a survey of the most common instrument software systems, which are primarily used for neutron but also for synchrotron and X-ray diffraction experiments.

2.4.1. Control and Data Acquisition Software

Caress

Caress [RS07, Sau07] is the primary control and data acquisition software for neutron scattering instruments at the Helmholtz-Centre Berlin for Materials and Energy (HZB) and also used at FRM-II near Munich. It is primarily a command driven architecture that controls about 130 different types of hardware components and is used at more than 17 instruments. *Caress* performs operations such as scans or the execution of scripts. It is a client server system with multiple hardware servers that publish a generic Common Object Request Bro-

ker Architecture (CORBA) interface. A graphical user interface extends the command line interface by setup and scan dialogs. Chapter 5.4 covers Caress in more detail.

Frack

Frack [Fle05] is a client server architecture that was developed at HZB in 2005 for advanced instrument control and calibration. The client is a pure platform independent Java application that owns a wizard guided fail safe swing user interface. Frack uses a simple proprietary XML based communication protocol for the communication with Frack Servers such as HardFrack-CARESS for the communication with CARESS or HardFrack-CAM for the communication with a visual sample edge detection system. Frack includes built-in modules for user management, instrument configuration, data analysis, data processing, instrument control and automatic generation of high quality PDF, PS and LaTeXpapers with integrated graphics.

TACO

TACO [Unr02, Neu98, GKM+09] is an object oriented control system that has been developed at ESRF Grenoble. It offers access to different types of electronic equipment by transparent network devices, which allow the addressing of device values by common physical units. The protocol used for calls between hardware servers and clients is Sun-RPC, which enables accessing services in a platform independent way. All features of devices are realized using device classes, written in C/C++ and are represented in form of Device Objects. Access to implemented device commands from within the program is simplified by a C/C++ API. It has been ported to the most common operating systems and can access CORBA, Java, TCL/Tk as well as ASCII based communication protocols, Labview via ASCII, OLE2 (Windows) and SPEC.

TANGO

TANGO [KAEJ98, tan09, CAK+99, CGK+01, GTP+03, GBS+07, GBS+07] is the successor of TACO and, as was its ancestor, is an object oriented distributed control system. It is also based on the device server concept but introduces a more advanced object model than TANGO. The Open Source system is actively developed by the institutes Alba[alb09], Desy

[des09], Elettra [ele09], ESRF [esr09] and Soleil [sol09]. In contrast to TACO, TANGO used CORBA as communication solution and so is programming language independent. TANGO clients and servers can currently be written in C++, Java and Python. TANGO objects as well as TACO objects are instances of a central Device class that defines the name properties, attributes, executable commands and events [Mit03]. Several services such as MySQL database access, naming services, archiving of historical data and a timing source are provided as well as a high level API that hides networking details, allows object browsing and discovery and offers several security features. Most clients are written in Java and handle configuration, monitoring and control tasks. Web based clients for TANGO devices can be developed with help of the tool Canone. One further step is the adoption of the CORBA Component Model to allow better mappings between TANGO components and Enterprise Java Beans (EJB).

NICOS

NICOS [Neu98, Unr02, KN02, GKM+09] is a network based instrument control system that includes the *Qt*[17]-based *NICOS Client*, a *NICOS Server* and the *NICOS Methods*. It is developed at the FRM-II in Garching, Germany and used as a front-end for TACO. The client and server are both written in Python and enable the writing nearly of all types of Python scripts using a client side editor and executing them on the server side. All devices are represented by object oriented classes and made available by the NICOS Methods package. NICOS introduces powerful enhancements and instrument specific extensions that allow all supported hardware setups to be deal with in a simplified and standardized way. A configuration editor and debugging capabilities as well as monitoring capabilities top off the range.

EPICS

The Experimental Physics and Industrial Control System (EPICS) [DKK91, DHK+93, Lew00, Hil95, KAJ+97a, TBO03, Car95] is a collection of software tools that work together and create a control and data acquisition software for experimental physics. It uses a dis-

[17]Qt is a well accepted API that inter alia allows to quickly build cross-platform UIs using *C++*.

tributed database, which registers functional subsystems like data acquisition systems, supervisory control, closed loop systems, archiving or alarm management. EPICS is built on a software communication bus that allows distributed process control. The realization of state machines is achieved by a control system that is modelled by means of a sequential control language. A timing system helps the triggering of components depending on given timing demands. All registered devices are readable as process variables within entire subnets via broadcast UDP packages. Hardware servers extend templates that are written in C with help of API functionality for interface declarations, data type definitions or timing. They are called Input/Output Controller (IOC)[KAJ+97b]. A special RECORD [SAK95] defines the process variables and triggering for the external access to components. Channel Access Clients for EPICS allow to remotely access devices and can be realized in different languages. EPICS is primary developed at the Argonne National Laboratory (ANL) [anl] in the USA and is in regular use at the synchrotron facility of the Helmholtz-Centre Berlin for Materials and Energy (HZB) [hzb] in Germany, at the Paul Scherrer Institut (PSI) [psi] Switzerland and currently introduced at Deutsches Elektronen-Synchrotron (DESY) [des09] in Hamburg. EPICS comes with the most comprehensive documentation library including manuals, tutorials and videos.

DANSE

DANSE [FAA, dana] is one of the latest well planned software projects on distributed **D**ata **A**nalysis for **N**eutron **S**cattering **E**xperiments. The development of DANSE started almost at the same time as that of Open Inspire. The project goals are 1) to build a software that enables new and more sophisticated science to be performed with Neutron scattering experiments, 2) make the analysis of data easier for all scientists and 3) to provide a robust software infrastructure that can be maintained in the future [danb]. The DANSE architecture is based on the data flow paradigm which connects reusable software components on different computers using data streams. The framework is *Open Source* and uses the language *Python* to enable the insertion of new procedures easily. Users are able to direct their processing in a web browser by accessing a central server. The user interface, that is consistent for all neutron instruments is intended to be easily operated. DANSE is language independent, can

integrate different kinds of commercial and non commercial software products and supports data storage with NeXus. Instrument scientists are able to use their scientific knowledge in programming DANSE modules while being shielded from the rest of the complex software. This decoupling is achieved by a property table that converts property strings to native representations. Components exchange data via ports and making it available to its core and vice versa. Each module decodes its incoming and encodes its outgoing DANSE-XML-stream depending on the current programming language, which ensures the decoupling of user interface and components. All components are available by a web portal using a standard browser and can be wired together with a Labview-like graphical user interface. DANSE is the reviewed software project with the highest funding. It is primary developed for the Spallation Neutron Source (SNS) in Oak Ridge Tennessee. The *National Science Foundation* originally awarded M$11.97 to the California Institute of Technology for computer software to analyze Neutron scattering experiments and develop best fitting software.

SICS

SICS [sic, HKM97, KW00] is an acronym for SINQ Instrument Control System and developed at the Paul Scherrer Institut (PSI) in Switzerland. Its primary use is the control of various instruments at the Swiss Spallation Neutron Source SINQ [sin] but in the meantime it also controls instruments at other institutes such as the Australian Nuclear Science and Technology Organisation (ANSTO) [ans] or South African Nuclear Energy Corporation (NECSA) [nec]. SICS is a client server system with one server and multiple clients that implement the SICS user interface. User interfaces are written in Tcl/TK [Ous94], ANSI-C and currently recoded in Java to provide a maximum platform independency. The core of a SICS system is the server, which is written in C but anyhow makes use of Object Oriented design principles. It runs on UNIX based systems and can be subdivided into a kernel, object database and interpreter. The kernel is the core of the system and responsible for the client access and general control. The objects are software modules that represent hardware devices, commands, measurement strategies and data storage. The last part is the interpreter that allows to issuing of commands to SICS objects that can be delivered as user commands from a command line. A TCP/IP terminal server provides access to the hardware via RS232, detector data is

read via TCP/IP network. The used network protocol between client and server is a simple ASCII command protocol through telnet or plain sockets. SICS is shipped with support for the latest version of the NeXus-dataformat since its maintainer is at the same time a founder of NeXus.

GumTree

The Graphical User interface for Multiple and Time Resolved Experiments (GumTree) [18] is primarily developed by a group of software specialists at the Australian Nuclear Science and Technology Organisation (ANSTO) [ans] but also released in different editions for instance at the PSI [KZH+08]. It is one of the most matured Integrated Scientific Experiment Environments (ISEE) due to the consequent use of latest software engineering techniques such as Agile Development[19] [HLH+04, BBvB+01, Coc02] including eXtreme Programming (XP) practices, which enables fast parallel development with frequent synchronization points instead of sequential development processes. GumTree evolved as a result of a requirement analysis process for neutron and X-Ray instrument software at ANSTO in 2002. Interviews, surveys and workshops were used to collect wishes and requirements of users and instrument responsible in different institutes. The result was the usage of a combination of SICS and Tango for hardware control, ISAW for visualization and GumTree as a client for superordinate control, data acquisition, visualization system for live and offline data and a workbench for data reduction and analysis. The developers were one of the first that analyzed similarities of common control systems architectures and proposed the Grand Unified Model for Control and Analysis Systems (GUM) [GH], which compares and defines several standards for scientific data acquisition systems. The Grand Unified Model for Control and Analysis Systems (GUM) states *"The control and analysis parts of a scientific experiment must be treated as part of one system with input and output being readily exchanged between all parts of the system. There must be a single integrated graphical user interface from which all aspects of the control and analysis system can be accessed. There is a basic set of building blocks that*

[18] GumTree related publications: [gum, LGFH06, LHG+06, RHH+06, HG, BK06, GSN06, GM07, CT, GH, BBvB+01, Coc02]

[19] "Agile Software Development emphasizes continuous adaptation to the project goals, developer skills, and personal interaction, over the traditional documentation bound, definitive plan and process driven practices." [HLH+04]

all control and analysis systems should have. All building blocks should have a well defined interface." Building blocks can be device drivers, device servers, network protocols, database, scanner, sequencer, batch, command line and graphical user interface, data file formats, analysis servers, analysis sequencer and data visualization. While the concrete realization of these blocks is up to the developer, the GUM defines an abstraction layer that can be interfaced by frameworks that are built to support multiple control systems. The components in GumTree are realized in form of lazy instantiated OSGi packages, handled by the Eclipse RCP. Components are available for the control systems EPICS, TANGO and SICS, the data formats, NeXus, HDF, Excel, CVS and XML as well as the visualization engines OPENGL, ISAW, Ptplot, VTK. Furthermore a Data Reduction and Analysis Module (DRA) [RHH+06] allows data reduction, correction and basic analysis while running experiments.

ACS

The Advanced Control System (ACS) is an upgrade[20] [KPP+] to the *ANKA Control System*, which has been used until 2000 to operate instruments at the synchrotron radiation light source ANKA [ank, PZS+02] in Karlsruhe, Germany. The same framework is also used in the ALMA Common Software (ACS) [CZ+05, SFS04] at the European Southern Observatory (ESO) [eso] and developed by the independent spin-off company Cosylab. ACS is designed in form of a three-tier architecture using a visualization layer with control GUI, a process control layer with accelerator objects and a fieldbus layer with devices. The visualization layer is realized by a rich set of JavaBeans which are part of the Abeans [KBC+, VKK−, DKO+] library, which has been released in 1999 as part of the *ANKA Control System*. The CosyBeans subset, which provides the widgets is today independently usable and used by other systems such as the Control System Studio (CSS). The process control layer consists of device servers, which were formerly implemented along the guidelines of the TACO system but in the meantime refactored to a new device server design [CJS+04]. These C++ based servers use CORBA to publish objects of all controllable devices. The third layer is the field bus layer, which is realized in form of a LonWorks field bus and specific hardware that allows to access the devices via device objects. The system provides an XML-based

[20]The upgrade was necessary to substitute the old CORBA implementation by an Open Source variant

configuration database and features such as synchronous and asynchronous communication, fail safe monitors and alarms, run-time name/location resolution, archiving as well as error handling and logging capabilities. An additional feature is the built-in management, which enables centralized control over processes with commands such as start/stop/reload, send message or disconnect client.

CSS

The Control System Studio (CSS) [css, HCG+] is a software project at DESY in Hamburg, Germany. It is like GumTree an Eclipse based Rich Client Platform (RCP) whose task is to provide a consistent and platform independent user interface for instruments that were before operated by several EPICS clients. The different look and feel and way of data handling of clients complicated the usability and data exchange, which lead to a common platform for developers with a management infrastructure and a centralized connection to external data sources such as JDBC-databases, JMS- or LDAP-servers. CSS uses OSGi and the Eclipse Extension Point mechanism to provide a pluggable design. The data is abstracted by the so called Data Access Layer (DAL), which allows to transparently access any control system protocol that implements the DAL and is so decoupled from EPICS. Several CSS applications are already available that allow browsing data [Kas] such as EPICS Process Variable (PV) tables, name spaces, alarm systems, provide rich sets of graphical widgets and a graphical editor for operator interface screens named SYNOPTIC [Kas07]. Data is persisted using the relational database Oracle and incorporate into different plug-ins [Pur]. Control widgets and visualization of live data is realized using the CosyBeans library [KBC+].

SScanSS

The Strain Scanning Simulation Software (SScanSS) [Fit, JSDE02, ZGA+09, JSED04] is an instrument software, developed for the diffractometer *ENGIN-X*[21] [OSJ+04, SDJE06] [ZGA+09] at ISIS [isi] (UK) but also installed at other neutron facilities including *KOWARI* at ANSTO, *NRSF2* at *OakRidge (USA)*, *VULCAN* at the SNS *(USA)* and *JEEP* at *DIAMOND (UK)*. It is written using the Interactive Data Language (IDL) [idl], an array-based sci-

[21] ENGIN-X is the successor of ENGIN [JEW97, DDE+04]

entific programming language for complex interactive mathematical and visualization tasks. A Graphical User Interface (GUI) helps to plan measurements using a virtual sample (obtained from laser scans or tomography data) and creates HDF5 and text based batch files, used to simulate or control the real instrument. It is possible to calculate neutron path lengths, required count times as well as predict potential collisions between specimen and the instrument. A compact mathematical framework has been developed to allow automatic positioning and orientation by robotic methods [JOP+08, JE07]. The IDL code is executed inside a free but commercial virtual machine that runs on all popular operating systems, while altering code is subject to users with valid IDL licenses.

NoMad

The software system NoMad [nom, iae] is a replacement of the dated software MAD and developed for instruments at the Institute Laue-Langevin (ILL) in Grenoble, France. It replaces the old undocumented fortran code of MAD with an object oriented software with a WYSIWYG[22] User Interface. All configuration tasks can be performed via drag and drop operations, even the definition of measurement batch processes. NoMad contains numerous predefined configurations that can be tailored to the users needs by setup dialogs. It is the first instrument system that is developed to be configurable by pen tablets right at the instrument and has a replay functionality that allows reproducing operations in the smallest detail. NoMad features a virtual simulation mode and is able to write data in the NeXus format and plain text. It is not an Open Source project.

SPEC

SPEC is commercial scientific software for UNIX that consists of packages for instrument control and data acquisition for X-Ray diffraction at synchrotrons. It has device control capabilities as well as visualization, a command interpreter, a programming language with a C-like syntax and an own data format. SPEC can be used to access devices at VME, CAMAC, GPIB, RS-232, PC I/O ports and socket I/O. It is developed by Certified Scientific Software (CSS) [spe] at Cambridge USA.

[22] What You See Is What You Get (WYSIWYG)

LabVIEW

The Laboratory Virtual Instrumentation Engineering Workbench (LabVIEW) [lab09] is a well known commercial development environment and platform for control and measurement systems. It is developed by National Instruments and has been continually improved and extended during the last 30 years. It is widely used in materials science for the control of instrument [DS06] and sample environment and basis for miscellaneous systems such as the control system of ENGIN-X. An essential difference to other systems is the introduction of the visual programming language *G*, which allows modeling programs in form of component boxes and communication links. The use of the dataflow paradigm combined with fine grained visual modeling allows to create nearly everything that is usually done on a textual basis in conventional programming languages. LabVIEW contains an editor, compiler and a debugger, which allows setting break-points, stepping through the structures and animating the program flow. LabVIEW programs are called Virtual Instrument (VI) and consist of the graphical block based programcode and a frontpanel. The frontpanel allows to model sophisticated user interfaces with help of a rich set of graphical widgets [JJ06]. LabVIEW code is compiled, has a comparable performance to other high level languages and can be executed on Windows, UNIX/Linux, and Mac OSX. A large number of tutorials, an active community [lab] and many books help starting with LabVIEW [TK06] as well as solving complex tasks [SS07].

2.4.2. Data Analysis, Processing and Visualization Software

BEAN

BEAN [FHSL99] is a program package for one-dimensional neutron spectra. It is used as replacement for different proprietary software packages at Berlin Neutron Scattering Center (BENSC) to provide a consistent data analysis system. It is designed as a 3-tier architecture. BEAN itself runs on a UNIX based Application server and has a connection to a client and database tier. The client is a simple workstation that is able to emulate X11-terminals and the used database is a NFS server. BEAN utilizes PV-WAVE for visualization and has a command line that extends the PV-WAVE command language with object oriented data.

The main task of BEAN is reading spectra with different data formats from the instrument, perform fitting operations and visualize data.

Open GENIE

Open GENIE [ope, ALS+a, ALS−b] is the successor of GENIE, a software to display and analyze neutron scattering data of instruments at the ISIS [isi] spallation neutron source, UK. Beside capabilities for data reading, analysis and visualization it provides specific features for neutron experiments as well as a scripting control, which is used at instruments such as ENGIN-X [OSJ+04, CAMS02]. It is distributed under the GNU Public License (GPL) and available for Windows, UNIX/Linux, and Mac OS. One strength is the ability to interpret multiple data formats such as simple plain text or even complex NeXus sources. Open GENIE incorporates technologies such as C and Fortran based extensible mechanisms, a Smalltalk Interpreter, a Tcl/Tk interface as well as PGPlot. A DCOM / ActiveX interface enables communicating with Windows programs such as LabView or Excel and yet another interface allows accessing and visualizing Matlab data.

ISAW

The Integrated Spectral Analysis Workbench (ISAW) [isa, CMM+02, CWH+01, CWH+00a, WMM+04] is a software project of the former Intense Pulsed Neutron Source (IPNS) [ipn] and today also used in other software such as GumTree. It is network-based and used to read, manipulate, visualize and save neutron scattering data. The choice of Java makes ISAW platform independent and even usable in browsers. It is very flexible due to its object oriented modular architecture, which makes it easier to maintain and add features [CWH+00b]. A GUI allows loading and manipulating multiple spectra from different *runs* and allow operations such as merging, summing, axes conversions, sorting and display the data as images or plots [GM07]. The support of NeXus allows to exchange measurement data between different software solutions and perform a comprehensive number of mathematical operations on data measured at miscellaneous instruments [CWH+00b, TBW+06].

DAVE

The Data Analysis and Visualization Environment (DAVE) [dav] is a development of the Center for Neutron Research at the National Institute of Standards and Technology (NIST) [nisb], US. DAVE is an IDL[23] based integrated environment for the reduction, visualization and analysis of inelastic neutron scattering data. It is a user friendly tool, which helps scientists to interpret data from neutron scattering experiments and comes with support for several spectrometers at the NIST Center for Neutron Research (NCNR) and Paul Scherrer Institut (PSI). A binary executable is freely available for Windows, Linux, and MacOSX. The open source code is subject to users with a valid IDL development license.

Mantid

The Manipulation and Analysis Toolkit for ISIS Data (Mantid) [man, DTT+08] is an application framework for high-performance computing on neutron data. The project, that is still in development, will provide an extensible framework with common services, algorithms and data objects and can be extended by plugins for specialized functionality. It is planned to be easily extensible by scientists and users, freely redistributable and should provide scripting, visualization and data transformation mechanisms as well as the support for virtual instrument geometries. The C++ and Python based data reduction and analysis tool Mantid is released under the Gnu GPL and available for Windows and Linux. It provides a Matlab interface and is NeXus capable [Fow08].

TVtueb / TVnexus

TVtueb [KBF+00, CSH+02, HHS03] is scientific software for data analysis and visualization of one- or two-dimensional datasets that are read from different types of data files. It has been developed at the Helmholtz-Centre Berlin for Materials and Energy (HZB) and is still in use for data reduction, analysis and visualization tasks of stress and strain measurements performed at the BENSC and FRM-II. TVtueb can directly read Caress data files and perform operations such as auto-indexing, determination of intensity and position of peaks,

[23] The SScanSS paragraph on page 39 contains a short survey of the Interactive Data Language (IDL)

transformation into reciprocal space, background analysis, intersections, three-dimensional fitting algorithms, correction of PSD-efficiency and the comfortable plotting of parameters. TVNexus [tvn] emerged from the codebase of TVtueb but has been extensively refactored to handle the data of new instruments such as the *E2* at the HZB. It comes with support for NeXus and can handle very large datafiles due to its new 64 bit design. TVtueb and TVnexus are both closed source applications and exclusively published for Windows systems.

2.4.3. Instrument and Experiment Simulation Software

One technique that became more and more accepted during the last years is the simulation of material scientific problems. Several years ago the computing-power was too weak to process complex experiments and instrument calculations but in the meantime it is possible to even simulate comprehensive experiments and instrument setups in acceptable time without the need for expensive computing clusters. For the development of modern instrument software these simulation techniques can help in three ways. Harnessing simulations to concrete experiments helps selecting areas of interest in advance and so reduces beam-time. While an experiment is running a simultaneous simulation can be used to precalculate the next reasonable measurement step without wasting beam-time. When the experiment finished, non-measurable data[24] can be simulated to get an idea how the measured data could look like and achieve a broader survey to the results.

The Monte Carlo method

The so-called Monte Carlo algorithm [GS96] is a well-established stochastic simulation method that is used in many scientific areas. Its name is a reference to a famous Casino in Monaco because of the analogous randomness and repetitive nature of the algorithm. It is often used for simulation of physical or mathematical systems, where an exact result with a deterministic approach is not feasible.

In Neutron Scattering environments, Monte Carlo simulations are primarily used to optimize the design of beam-lines[25].

[24] For example areas that can not be measured due to the sample or instrument geometry
[25] Beamline Simulation Publications: [AdRF04, ALTK+04, ALN01, BPHLB06, DSHT99, NKM95, SB78]

Starting from the simulation of the neutron sources, described by the neutron energy distribution and its geometrical shape, single neutrons starting at the source with randomly calculated velocity and direction are being sent on their way into beam tubes, instrument shielding and optical components. The interaction of each neutron with the instrument components in their flight path is calculated and the neutron direction and wavelength is changed respectively. Finally the neutron is absorbed by shielding components or detected by the virtual instrument detector. Calculations for millions of neutrons can be done very rapidly on today's computer clusters. Generally the assumed characteristics of the sample on the center of the sample table are very simple because these calculations are not done to simulate specific experiments but only to determine/optimize instrument resolutions and signal-to-noise ratios. Therefore the available sample descriptions normally don't contain information about absorption, complex geometry, orientation distribution, crystal structure of individual phases and so on. For this type of work several Monte Carlo Simulation frameworks are available on the science market.

Simulation Frameworks

Commonly used simulation systems in this area are the frameworks McStas, VITESS, NISP, IDEAS and Restrax. All of these tools deliver feasible and comparable results [SDF+02] and only differ in their scope of operation. Here the frameworks differ in their usability, user-interfaces, scripting and configuration capabilities as well as the range of prebuilt modules. For most applications it is sufficient to select the one framework that either already comes with a fitting set of functionality such as prebuilt modules for specific neutron optics or to select the framework with the preferred usability. Sometimes simulation systems such as McStas and Vitess [RI07, UO08] or McStas and Restrax [WSF+00, WSF+02] are used in combination to compare the results or profit from the combination of the strength of a specific environment.

McStas

McStas [mcsa] is one of the most popular neutron ray-tracing simulation tools. It is based on a meta language especially designed for neutron simulation experiments but in the mean-

time also available for X-ray experiments [mcx]. A component-based approach lets the user select, arrange and parameterize components in the McStas meta-language, which is automatically translated to ANSI-C Code. This technique allows the creation of a simulation executable, which can be run with high performance even on systems, where no McStas is installed. McStas supports both pulsed and continuous sources and is able to simulate all kinds of neutron scattering instruments such as diffractometers, spectrometers, reflectometers or small-angle scattering instruments. The McStas Library currently includes around 100 components; VITESS components can be included, too. A big advantage of McStas is the large number of different data formats that can be generated including PGPLOT, Matlab, Scilab, VRML/HTML, HDF/NeXus, XML/NeXus, Octave and IDL. McStas is actively supported by the RisøNational Laboratory[ris], the Institute Laue-Langevin (ILL) [ill], the Paul Scherrer Institut (PSI) [psi] and the Niels Bohr Institutet Kobenhavn (NBI) [nbi] and is freely available under the terms of the GPL. The package is well tested, reliable and available for Linux, Mac OS and Windows. It is comprehensively documented and supported by an active community. The large acceptance of McStas becomes obvious by the large number of publications[26] [mcsb] about successful simulations at the RNL, ESS, ILL, FRM-II, PSI and ANSTO.

VITESS

The Virtual Instrumentation Tool for Neutron Scattering at Pulsed and Continuous Sources (VITESS) is the second widely used monte-carlo-based simulation environment. "VITESS has been partly supported by the SCANS network (FP5) and was supported by the NMI3-MCNSI Network (FP6) within the Research Infrastructures Activities of the Research and Technology Development Programme of the European Commission." [vita] The free software is maintained and permanently extended by the HZB, ILL, PSI, JINR [jin] and IFE [ife]. The developers of VITESS attached importance in providing a full featured Graphical User Interface (GUI) for the setup of simulations, data visualization and analysis. It is, as well as McStas, built in a modular way and can be used for the simulation of nearly all types of beamline installations including the neutron source, guides, choppers, polarizers, monochromators, analyzers and

[26] McStas Simulation Publications: [ALN, GFKW07, GOP07, Gra07, GDVL+08, HSG+07, KLM+02, LWF, LN99, LNTL00, mcsa, NL00, SBFH04, VGF+03, WKF+, WFL04, WFL+05, ZSK+06]

even the sample. The internal design is built on a modular description architecture based on building blocks and pipe based communication mechanisms [Fro]. VITESS is the simulation system with the second largest number of publications[27].

NISP

The Neutron Instrument Simulation Package (NISP)[28] queues up in the list of montecarlo based neutron-ray-tracing packages. The package is developed in Los Alamos and consists in a Graphical User Interface for Windows, the Fortran Monte Carlo Library MCLIB [See95], the simulation executor *MC_Run* and the Windows based results viewer *See_MC_Data.* NISP can perform complete source-to-detector simulations of neutrons, is well documented but has a smaller community than McStas or VITESS.

Other M.C. based Neutron Diffraction Packages

Beside the well-established simulation packages introduced in the previous sections some further packages are worth to be mentioned. These packages are used rather infrequently for instrument optimizations but can be considered for some situations in which they provide additional functionality or simply promise a better result for a specific situation.

The first tool that belong to this group is RESTRAX and its modification SIMRES [res][29]. It can be used for the planning, simulation and analysis of neutron scattering instruments. The code is reliable and creates output comparable with the output of McStas or VITESS [SDF+02].

The second mentionable Neutron Scattering Simulation package is IDEAS [LW02, LWR+02], which has already been applied to instrumentation projects at the ORNL, HZB, FRM-II and BNL. IDEAS is developed for Windows Systems and completely configurable using its Graphical User Interface. It can be by extended either in the source code or by DLLs using code in C or FORTRAN.

[27] VITESS Publications: [LZM+, LGWM05, MZA08, vita, WZS+00, WZSM00, ZLM02, ZLM+04]
[28] NISP Publications: [nisa, spi99, SDTH00, SD04]
[29] Restrax Publications: [ŠK06, WSF+00, WSF+02]

Finally the tool MCNPX [GFKW07], which has been developed at the Los Alamos National Lab (LANL) [lan], differs from all other packages. "MCNPX is a general-purpose Monte Carlo radiation transport code for modeling the interaction of radiation with everything". [mcn] The difference is that it is not specially optimized for Neutron Diffraction Applications but allows calculations beyond the scope of all other packages. So it is used for nuclear medicine, nuclear safeguards, accelerator applications, homeland security, nuclear criticality, and much more. MCNPX is written in Fortran 90 and runnable on UNIX and Linux systems.

2.4.4. Data Exchange and Storage

Whenever automated experiments are performed, collected data needs to be stored for further processing or archival storage. Focusing on the last three decades it is observable that the demands on a suitable dataformat are gradually shifting. While aged solutions are primarily focused on saving disk space and a providing high-performance algorithms, modern solutions also aspire to afford a highest possible exchangeability.

The reviewed facts about common software system show that nearly all older systems come with a proprietary data format. For short term data storage, this is acceptable since data measured by different systems can significantly differ and as much data should be stored in the best fitting way.

Short Term Storage

Short term storage mechanisms are either necessary when the experiment generated data exceeds the amount of volatile memory or the complexity of data does not allow to store it in a common data format. One automatic way to deal with this problem is using the OSs swap mechanism, which automatically caches data that does not fit in memory to the harddisk. For time-critical tasks this mechanism has several disadvantages since it is not predictable when data is stored and which data is stored. Here the access, read and write times significantly differ depending on the amount and speed of RAM as well as the type of swap medium such as a HD or SSD. Furthermore the addressable memory depends on the hardware and OS architecture and is for 32 bit systems limited to 4GB, which is not enough for large experiment data, such as the detector output of instrument *E2* at BER-II.

A different setup that solves these problems is storing the data on demand using persistence mechanisms. While most instrument systems persist their data to proprietary binary files, new architectures in different areas make use of high level frameworks such as *Hibernate*[30]. These framework often provide different standardized ways to directly serialize objects to files or databases. Naturally these files can also be manually stored in relation databases such as *MySQL*, *PostgreSQL*, *Oracle* etc. using SQL-APIs like JDBC.

Long Term Storage

Beside the internal data storage, that is more a technical issue, another important topic is the long term storage of data. Long term storage in this context means the storage of data using a standardized format, which can be read by independent tools even after many years. Here the ongoing globalization in diffraction experiments shows that there is a strong need to compare measured data. A way to achieve this is to agree upon standards and store the measurement results in a common data format. This allows to open and compare experiment data obtained from multiple instruments by all tools that support this unique standard. Here the data differs significantly from the short term data. While the *short-term-data* is equipped with verbose instrument specific information, for the longer stored data, usually data-reduction algorithms can be applied. The remaining relevant data finally needs to be saved in combination with meta information that allows to recognize data structures in a consistent way. To reduce storage costs and to simplify file transfers it is useful to choose an output format which supports compression.

The NeXus Format

Many scientific branches draw upon one or more common data formats to store data combined with meta information. In the area of neutron, x-ray, and muon science the NeXus dataformat has been established as a widely used standard.

The Data Format for Neutron, X-Ray & Muon Science (NeXus) is an open file format that has been standardized by the NeXus International Advisory Committee (NIAC) [nia09], a group of programmers and scientists representing major scientific facilities in Europe, Asia,

[30] *Hibernate* is an Object Relational Mapping library, which is e.g. able to map *Java* classes to database tables

Australia, and North America. The committee meets at least once every year and deals with questions such as the establishment of policies concerning definitions, use and promotion of the NeXus format, insurance of a sufficiently complete and clear dataformat as well as the discussion of new definitions, interfaces and API functionality.

The NeXus architecture is composed of four components [nex09] :

1. Design principles and guidelines that describe how and where data is to be stored
2. Instrument descriptions that form unique templates for portable instrument data
3. An access layer to lower-level file formats that store NeXus files on physical media
4. An API with a set of routines that allow to read and write NeXus files

Design principles

The NeXus-design-principles describe the general assembly of NeXus-files without fixing the underlying file format. A NeXus-file can be regarded as a file system in which the two NeXus main entities *data groups* and *data items* form a tree. *Data groups* represent the branches and can contain *data items* or further groups. *Data items* are represented by so called Scientific Datasets (SDSs) and can be scalar values or multidimensional arrays with character-, integer- or float-types of various sizes[31]. In contrast to plain filesystems NeXus additionally introduces some sort of meta information that help to identify stored data. *Data items* are tagged for this with attributes in form of key-value pairs that contain extra information such as data units. Groups do not have attributes but are both, identified by a name and an additional class identifier that explicitly defines the sort of data stored in a group.

Illustration 2.8 shows a simplified example of a NeXus tree structure. Here it is visible that each group has a dedicated NeXus-class that starts with *NX* and is printed in brackets under the group name. Each file only contains one *NXroot* but can contain as many *NXentry*[32] elements as needed. All further guidelines that describe which and how many entries of *data items* and *data groups* can be stored in a group are defined by XML based meta description files [nexc]. This also includes the allowed types of attributes that can be assigned to *data*

[31] Allowed sizes are 1,2,4 and 8 bytes
[32] An NXentry contains all data for one single scan or measurement

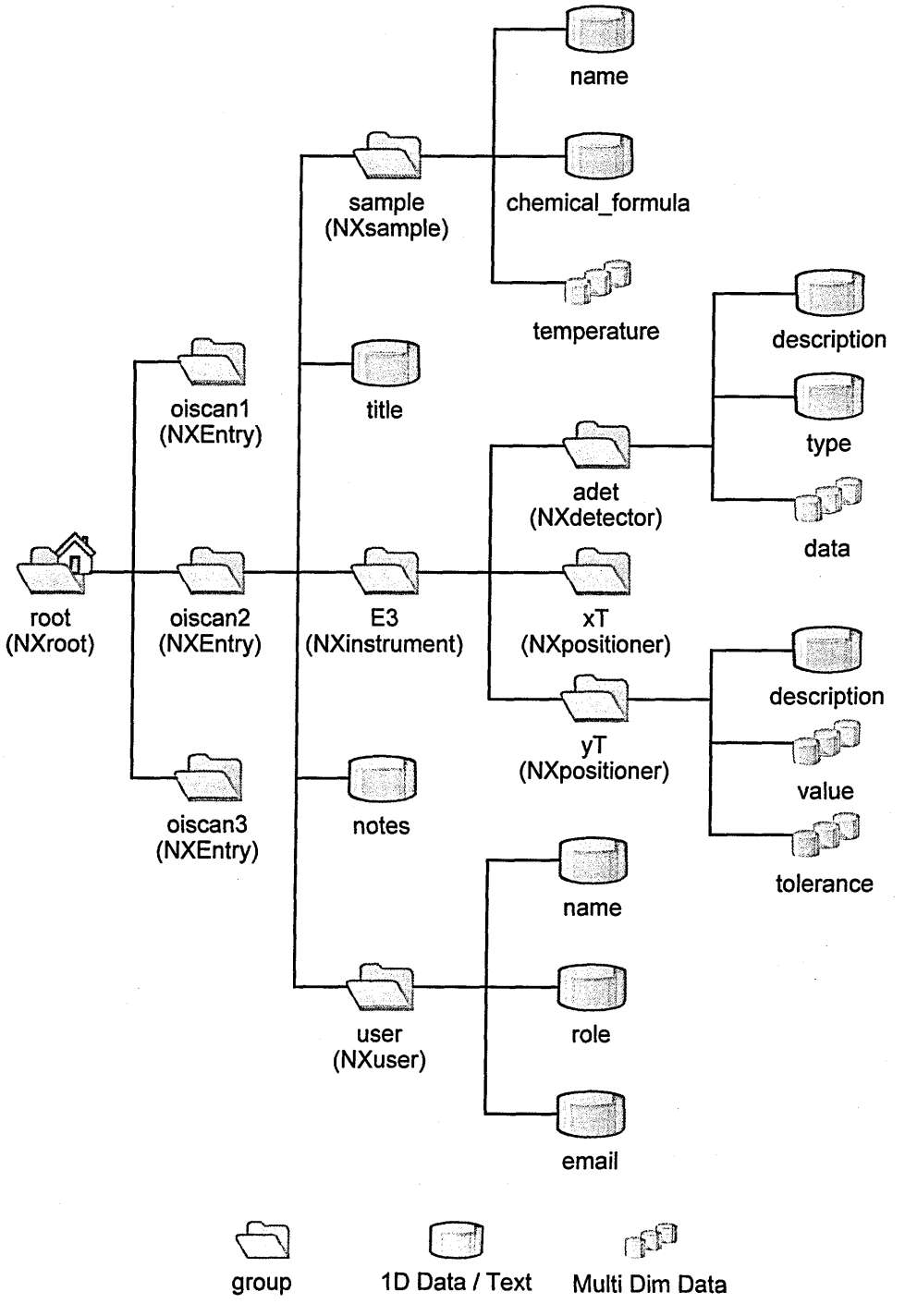


Figure 2.8.: *Illustration of a NeXus Tree Structure*

items. It is important to mention that a tree may be extended by own proprietary groups and items. The drawback is that independent software can no longer recognize the type of the proprietary stored data.

Instrument Descriptions

The previous chapter illustrated that NeXus files are equipped with meta data that help to identify stored information in a human and machine readable way. In some situations this is not sufficient since mandatory relations between stored data for specific experiment or instrument scenarios are not addressed. NeXus defines a set of instrument definitions that solve the problem by precisely defining the format for particular instrument types. These definitions are being formalized as XML-files and describe all data groups and items including a flag if they are optional or mandatory. The existence of these definitions allow to develop much more portable applications for data analysis and visualization that can rely on complete and valid experiment and instrument data.[nexb]

Low Level NeXus File Formats

Beside the definition of structures and guidelines a further important point is the storage on physical media. Here NeXus separates the organization of the data from the concrete storage format. This means that NeXus allows the storage of data using different fileformats that ideally support tree based structures. The three directly supported file formats are XML, HDF4 and HDF5. While XML has the major advantage that it is directly human readable but also simply parsable by a multitude of free XML parser libraries, its drawback is the high consumption of storage space due to its verbose meta information. HDF5 on the other hand allows high compression and fast data handling but requires special HDF capable libraries that are not available for all platforms[33]. The support of HDF4 is more a compatibility decision and only supports a subset of the newer HDF5 functionality[34].

[33] One example is the absence of a 100% native HDF5 java library and even missing 64 Bit support when falling back to the Java Native Interface (JNI) based HDF5 implementations

[34] HDF4 in particular lacks the support for multidimensional compound datatypes, is not thread safe, does not natively support NeXus trees and has size restrictions [hdfa]

The NeXus API

The final part of this NeXus disquisition is the NeXus Application Programming Interface. This API is an integral part of NeXus and provides a unique interface that allows to read, write and create NeXus files in XML, HDF4 and HDF5. The advantage of this abstraction layer is that the same routines are used for all three formats. Switching between them is possible with minimal effort. The API is available for C, Fortran 77, Fortran 90, Java and IDL. Support for further languages such as Python is being planned.

The navigation through NeXus-files follows a state machine based approach. After opening the NeXus-file, a handle is used to store the current location and allows to navigate through the tree similar to the browsing of a file system using a Command Line Interface (CLI). Commands are provided to open, close and create groups, read and write data and related attributes as well as setting compression flags and the type of the underlying file format.

NeXus is in the meantime supported by several instrument control systems such as SICS, GumTree, NoMAD and DANSE, analysis and visualization systems such as ISAW, and Open GENIE, Mantid, TVnexus and simulations systems such as McStas. More detailed information about the usage of NeXus in Open Inspire is covered in chapter 5.2 on page 190.

CHAPTER

THREE

SOFTWARE REQUIREMENTS

3.1. Introduction

The previous chapter gave an overview of the scientific backgrounds, the instruments and research environment, measurement principles and the most commonly used software solutions.

This chapter now determines the requirements that are placed on an instrument system. This is realized using a Software Requirements Specification (SRS), which later serves as basis for all decisions during the development process.

For this specification, the author decided on a mix of different forms of description that have been shortened and adapted to the context of a PhD thesis. As a basis, the Guide for Developing System Requirements Specifications (IEEE 1233) and Recommended Practice for Software Requirements Specifications (IEEE 830) have been applied [SAF00]. The structuring is based on the *Volere - Requirements Specification Template* [RR09].

To determine the software requirements, a specification analysis and continuous enhancement has been carried out considering the following sources:

a) Anonymous interviews conducted with instrument responsibles, users, developers and decision makers at selected institutes[1].

b) Evaluation of the results of the Inspire Assessment, a questionnaire conducted in December 2007 in cooperation with the TUM, NECSA, PSI and ILL.

c) Analyses of functional and non-functional properties of the instrument systems introduced in chapter 2 including the analysis of source codes and system processes.

d) Analysis of the usual processes and operating experience at instruments and their environment (e.g. scans) regardless of the operated soft- and hardware.

e) Experiences and observations of other hardware and software systems with overlapping functionality (e.g. in other scientific or industrial disciplines).

f) Consideration of similar programs or libraries or dependencies used by other systems.

The complete Requirements Analysis includes the identification of the abstract *Customer Requirements* using the sources *a* and *b* and their further development to *Development Requirements* by means of source *c* to *f*. [2]

In favor of readability and the scope of work the presented individual evaluation of all intermediate steps of the requirement analysis will be omitted. The separate determination of customer and development requirements are also not part of this document as the interviews were carried out anonymously[3].

However included are the results of the finished requirement analysis in forms of a reduced Software Requirements Specification (SRS). This includes the presentation of the *Business-*, *Product-* and *Process Requirements* and is divided into the three main sections *The Purpose of the Project*, *Functional Requirements* and *Non-Functional Requirements.*

The selected form of the SRS allows an overview of all requirements to be obtained, and allows their referencing during the design and implementation phase in order to evaluate the satisfaction of the requirements after completion of the development. The SRS is thereby

[1]Included Institutes: HZB (former HMI and BESSY), TUM, ANSTO, NECSA, PSI, ILL, DESY and OU
[2]For more information about Customer and Development Requirements read [CKM01]
[3]The *Inspire Assessment* was public.

not limited to Open Inspire but can be reused as basis specification for other instrument or control systems.

For the prototype of Open Inspire (which will be covered in 4), the SRS is indispensable in that it allows a systematic continuation of the development even beyond the completion of the prototype. Software companies organize extensive processes generally according the Capability Maturity Model Integration (CMMI) reference model [Tea02], which subdivides the quality of the software project organization in five "maturity levels". Already CMMI Level 2 *(Managed)* requires the existence of a SRS to handle the development processes.

3.2. Project Drivers

The first part of the SRS gives a short overview of the Project Effort, the Goals of the Development, introduces the Stakeholders that are confronted with the instrument system and summarizes the constraints of the development.

3.2.1. Background of the Development Effort

For many years more and more material scientific tasks in international institutes are assisted by a growing number of software systems. Initially simple process computers executed repetitive tasks to disburden personnel, increase the reproducibility and lower the errors of measurements. In the meantime these computers are replaced by several generations of less expensive, more powerful and easier to use computer systems that allow measurements that could before only be made under challenging conditions. Nearly all institutes took advantage of these new technologies and developed instrument systems as introduced in section 2.4.

Today's growing computer networking, distribution of tasks, the implementation of cloud computing and the global convergence of research institutions give more than ever a motivation to use these new concepts for the enhancement of existing systems and profit from new developments.

3.2.2. Goals of the Project

The goal of the work is to develop the concept of a universal and flexible Supervisory Control and Data Acquisition (SCADA) middleware that bridges the gap between established classic systems such as introduced in section 2.4 and a global cloud oriented service infrastructure. The software should therefore ease the loose coupling of already available and working classically designed software fragments and increase their strength in a global context.

The heart of the concept shall be a framework and platform that improves the collaboration between users and developers of instrument systems. This shall be done by providing means that allow the global publishing and sharing of binary software fragments and the composing of them to arbitrary instrument and experiment setups. The vision is that users can quickly compose experiment setups using prebuilt building blocks without the need for programming skills and that developers can share and reuse common functionality to reduce redundancy and save time. With sophisticated mechanisms for the coupling of heterogeneous Data Acquisition and Control (DAC) systems it should also help to discover common interfaces, test the compability of systems and flexibly define and test new interface standards.

3.2.3. The Stakeholders

The first step of the SRS comprises the identification of all involved stakeholders. The side of the client is represented by research institutions that offer experiment software to users. On the customer side are the experimenters and observers that use and maintain the system and the last group includes the developers that are responsible for the development of the core system, extensions and software tests.

The Client - Research Facilities

The system to be developed is not dedicated to one or only a few clients but freely usable by all facilities that offer controllable research or automation infrastructure to clients. For the investigation of the requirements and operating environments of potentially interested research institutes several instrument responsibles and developers have been interviewed.

Facilities with *instrument responsibles* that offered interest are the HZB (DE), FRM-II (DE), PSI (SW), NECSA (SA), ANSTO (AU), ILL (FR), DESY (DE) and HZDR (DE).

The research facilities select and provide the software and infrastructure for experiments, pay soft- and hardware maintainers as well as developers and schedule the access to instruments and the sample environment.

The Users

While research institutions play an important role as providers of infrastructure the *user*-group actively makes use of the actual system functionality. Users can be divided into observers, experimenters, configurators and administrators.

Observer

The *observer*-group is the group with the lowest privileges and includes all persons that have no active input privileges to the software but are enabled to query information from the system. Examples are the current position of an axis, the visualization of a detector, the state of hardware components or error messages.

One variant for a user interface accessible by *observers* is a public display directly at the instrument, which can be viewed by everyone. An other active variant may be a network-based remote monitoring using remote computers, tablets or mobile phones.

People which are belonging to this role are for example visitors that pass the instrument but also security guards or maintenance personnel that have no direct reference and access to the instrument but should be alarmed about possible malfunctions or hazards.

	Scientific Skills	Programming Skills	Instrument Setup	Electronics Setup
Level	□□□	□□□	□□□	□□□

Table 3.1.: *The Observer's Skills*

Table 3.1 shows the strength and knowledge required to be a member of the *Observer*-role.

([□□□] ≙ novice, [■□□] ≙ moderate experience, [■■□] ≙ experienced, [■■■] ≙ expert)

Experimenter

The largest and most important[4] group is the group of *experimenters* and includes the scientists conducting experiments on the instruments. An *experimenter* selects the appropriate experiment environment, configures measurement sequences and evaluates the measured data. They normally use the instrument only for the limited time of the experiment and then switches to the most promising instrument for the next experiment.

An *experimenter* usually has no or only limited programming skills but extensive scientific knowledge regarding the experiment and setup of the instrument.

	Scientific Skills	Programming Skills	Instrument Setup	Electronics Setup
Level	■■■	□□□	■■■	□□□

Table 3.2.: *The Experimenter's Skills*

Configurator

Configurators are responsible for the adaption of the software to specific instruments and experiment environments. Before the individual components of an instrument can be used it is usually necessary to configure the software with hardware dependent setting. Depending on the hardware this can for instance be *bus IDs*, *IO-*, *memory-* and IP addresses but also the *offsets* and *limits* of *positioners* or the number of *detector channels.* Depending on the software setup additional properties such as database configurations, the composition and Look and Feel (LnF) of the Graphical User Interface (GUI), preconfigured measurement sequences or storage paths are conceivable.

The *configurator* normally inherits all the privileges of the *experimenter.*

	Scientific Skills	Programming Skills	Instrument Setup	Electronics Setup
Level	■□□	□□□	■■■	■■■

Table 3.3.: *The Configurator's Skills*

[4]The experimenter is the user that primary makes use of the software. All other users work aims to make the experimenters measurements possible.

Administrator

A similar role to the *configurator*'s role is that of the *administrator*. This person is as well responsible for the system setup but has more privileges. While the *configurator* can only configure released properties of the installed instrument software, the *administrator* has unrestricted access to the whole system. Tasks of the *administrator* are the basic installation and maintenance of software, the rectification of errors, the installation of updates and the creation and management of users, groups and their privileges. Moreover he is responsible for the secure and reliable operation of the system. As superuser he inherits the rights of all other users.

	Scientific Skills	Programming Skills	Instrument Setup	Electronics Setup
Level	■□□	■□□	■■■	■■■

Table 3.4.: *The Administrator's Skills*

Developers

Beside the two categories *clients* and *users* the stakeholders can be differentiated into a third role *developers* that are responsible for the extension of the system. While software projects for small monolithic software normally do not differentiate between *System Designers*, *Component Developers* and *Software Tester*, it is indispensable for larger modular systems.

System Designer - Software Engineer

A person that is considered as a *System Designer* is a developer who is responsible for the abstract planning of the overall system. He is primary a *Software Engineer* whose work includes the requirement analysis, the object oriented analysis and modeling of the software system, the framework and interface design as well as the application of relevant *Design Patterns* and adherence to *Good Practices*. The results are usually model descriptions in form of UML diagrams. While implementing the software is actually not the work of the *Software Engineer* but the work of a *programmer*, providing runnable *reference models* in form of *prototypes* is not unusual and often helpful.

The practical part of this thesis' work mainly falls into the realm of *Software Engineering*.

	Scientific Skills	Programming Skills	Instrument Setup	Electronics Setup
Level	■□□	■■■	■■□	■■□

Table 3.5.: *The Developer's Skills*

Component Developer / Programmer

The group of *software developers* includes the programmers who develop the core software system against the *system model*[5] as well as the developers who write system extensions. *Core developers* mainly work at the project beginning and later on rarely make changes to the core system to prevent incompatibilities. *Component developers* on the other hand constantly extend a modular software system using component based extensions (plugins) that will later be configured by *Configurators* and used by *Experimenters.*

A software developer is usually *Administrator* for his own development and test system.

	Scientific Skills	Programming Skills	Instrument Setup	Electronics Setup
Level	■□□	■■□	■■□	■■□

Table 3.6.: *The Developer's Skills*

Software Tester

Software Tester are responsible for the test and verification of the software system. Their job is to setup test systems, create test cases (e.g using *unit tests*), analyze log files, profile the memory and realtime behaviour, perform code quality checks and verify if the software meets the given requirements and good practices. Projects that use Test-driven development (TDD) as development process first create tests and develop and refactor the functionality in short cycles until all tests succeed.

During the development of the software prototype in chapter 4, only a minimum of automated software tests will be applied since it would exceed the time constraints.[6]

[5] The definition of the *System Model* is the job of the *Software Engineer*
[6] This does not affect the mandatory manual tests that are performed before publishing a new release.

	Scientific Skills	Programming Skills	Instrument Setup	Electronics Setup
Level	■□□	■■□	■□□	■□□

Table 3.7.: *The Developer's Skills*

3.3. Project Constraints

3.3.1. Mandated Constraints

Mandated constraints are the essential constraints that must be met by the system. To keep it short and simple, it is presented in form of a summary rather than atomic requirements.

The Implementation Environment

The most important constraints refer to the implementation environment. A software prototype shall be developed that is able to control the neutron diffractometers *E3* and *E7* at the HZB. This consequently requires the system to be compatible with the instrument hard- and software, covered in chapter 2.3. To meet this requirement it must be possibly to query data from the instrument hardware and to gain a level of control over the hardware that allows to run scans. This includes the ability to configure and initialize hardware and to control and query *Caress*-controlled devices such as motors, encoders, detectors (single and area) as well as counters. The system must be executable at the instruments without needing to change any hardware. This is met when the software is runnable on the given instrument computers.

Partner or Collaborative Applications

The software used to control the hardware at the instruments is *Caress*. To control the existing devices without changing them it is necessary that the new system is able to communicate with them in the same way as *Caress* does. To achieve this, the system must come with its own implementation of the *Caress* protocols to allow the direct communication with the *Caress*-hardware-servers.

Beside the control it is also necessary to save data in a generally accepted data format. The NeXus-format, which has been covered in section 2.4.4 on page 49, is the standard format for

the storage of neutron diffraction data. The system shall then come with support to write measurement results using the *NeXus*-format.

With respect of the measurements it should be possible to run complex scans with minimum configuration effort. The tool SScanSS, which has been covered on page 2.4.1, allows the quick 3-dimensional planning of scans and writes the scan paths to an output file. The system shall be able to read these scan configurations and drive the hardware according to the presets.

Finally it shall be possible to simulate measurements rather than driving real hardware. This allows setups to be tested and dry-runs to be made in order to anticipate how the real measurement results may look like or assist the real measurement with pre-simulated data. To perform simulations, the system should allow the integration with the tool McStas, which has been introduced on page 45 and is the selected standard simulation engine on *E3* and *E7*.

Solution Constraints

The solution must be flexible enough to be adapted to new environments and situations. The developed prototype shall run on *Neutron Diffractrometers* but the overall system must be easily adaptable to other instrument hardware and be usable for measurement and control tasks in other scientific or industrial disciplines.

While the prototype shall be developed for the *Diffractometers* at the HZB, the core system must be designed in a hardware independent way. This means that only a minor part of the system, that is responsible for the communication for the hardware, needs to be adapted to run the same experiment on hardware used at other institutes. The aim is to perform the same experiments with different motors, counter and detectors but get comparable results.

To justify the effort of developing the new software it must have a clear advantage over classical systems. It must improve the way experiments are run without introducing new limitations and it may even allow new types of experiments that are not possible with current systems. This shall be achieved by a platform that allows a more flexible configuration, a better extensibility and an easier maintenance without reducing the overall performance.

Advanced measurement features shall be easily addable to the system by an extension mechanism. The aim is here to provide intelligent measurement features that, for example allow measurement times to be calculated dynamically in response to actual sensor or simulation data.

Legal Constraints

Everyone shall be able to convince himself whether the system code is secure and does exactly what the user wants to do with the system. This can be done by analyzing the source code. A legal constraint is therefore that the source code is publicly available. This is met when the system is published under an OSI compliant open source license and free to download by everyone from the internet.

To keep the system free of charge and avoid interfering with other license restrictions it must be independent and decoupled from commercial software. If external libraries are required for the functionality of the core system, these must be free and under an *Open Source* license. To avoid restricting the system however it must be both permissible and possible to extend it even with commercial libraries and tools.

Schedule Constraints

The work is done in the context of a *PhD*-thesis so that the final software prototype must be available within the *PhD* schedule of 6 years. A preview of the software must be available after 2 years that is able to control the devices of the given environment to confirm the feasibility.

Software tests will be run on the same instruments used for the regular user measurements. This makes it necessary to schedule them in a way that does not conflict with measurement times of such experimenters.

3.3.2. Relevant Facts and Assumptions

Relevant Facts

The instrument network is separated from the rest of the network using a Virtual Local Area Network (VLAN), which makes it impossible to directly access the instrument computers from the HZB intranet or the internet. The only way to communicate with the instrument from the outside is using a SSH tunnel. Slow or faulty networks, which have been identified at several institute networks, and the overhead of the encrypted tunnel can cause bad or fluctuating network latencies and throughputs. These problems must be countered by fault-tolerant network communication mechanisms.

The instruments are located in radioactive environments with extensive underlying safety rules, which restrict users in entering the instrument environment. This makes it necessary to consider waiting times if the direct access to an instrument is required and presumes a design that is stable enough to run in unattended environments with non-permanent user access.

Business Rules

All neutron sources have maintenance times. The reactor at the HZB is for one week per month shut-down so that software tests can be performed that do not require neutrons.

Assumptions

The main system users belong to the group of experimenters. These scientists want to perform measurements and solve problems in a timely and focused manner, reach the aim without writing complex and comprehensive scripts and avoiding configurations that require internal system knowledge.

Globalization approaches make it necessary that experiment results that have been performed with instruments at different institutes are comparable. To make experiments comparable, new standards such as the NeXus format need to be defined. These help to share data and to interchange measurement workflows, device-interfaces or configuration files.

The performance of computers will continuously increase (Moore's Law [Sch97]). Future computer hardware will be able to perform more complex operations such as simulation calculations.

It can be assumed that software which is currently being developed will later run on higher performance hardware and will be designed to make use of more complex functionality.

3.4. Scope of the Work

3.4.1. The Current Situation

Many institutes worldwide carry out material scientific experiments using neutron diffraction techniques. A short introduction to these techniques was given in section 2.2 on page 19. The environment and the assembly of the instruments has been covered in 2.3 on page 24 and the most frequently used software systems for diffraction experiments have been summarized in section 2.4 on page 32.

Before a measurement is performed, the first step is to find the best fitting instrument for the specific research. After an adequate instrument has been found, a proposal needs to be submitted to the operator of the instrument that contains a detailed description of the experiment and the demanded sample environment. If the feasibility of the experiment can be confirmed and the proposal has been accepted, so called beam time will be assigned in which the experiment has to be performed.

Now the real work begins and both sides start with the preparation of the experiment and the environment. This work is an important factor for the overall success of the experiment and increasingly includes simulation techniques that help identifying problems prior to the real experiment. One part of the work is the preparation of the samples and the planning of the measurement the other one is the configuration of the sample environment and the software. All currently available experiment systems support the use of batch-files, which describe the single steps of a measurement required to run long experiments without intervention. After completion of the measurement a dataset with all relevant measurement values and parameters will be given to the experimenters. Depending on the format, the analysis of the

measurement results can be performed by free standard software or using special software such as *TVtueb*, which will be given to the experimenters.

Each measurement system only has limited configuration capabilities that suffice for the commonly used experiments. If custom measurement flows or non-supported hardware need to be used, adopting the software can lead to a considerably time-afford.

3.4.2. The Context of the Work

Due to the limited overall-time of a *PhD*-thesis, only a model shall be developed that can be used as the basis for further control and data acquisition tasks at different instruments.

The verification of the model shall be proven using a prototype which demonstrates the core-concepts of the model. For a selected set of important features only one use-case will be and implemented for demonstration purposes. Although it would be possible to use the prototype in other environments, the verification shall be limited to the environment at the HZB which has been introduced in section 2.3.

Performance-, memory- and other fine-tuning work as well as the preparation for hard real-time will not be part of the prototype development unless it is essential for the functionality.

3.5. Requirements

Requirements can be split into the categories *Functional-* and *Non-Functional-Requirements*. *Functional Requirements* describe *what* the system shall do and *Non-Functional-Requirements* *how* it shall do the work.

This thesis deals with the development of an enhanced system, which improves the workflow of experimenters, administrators and developers, provides more flexible experiment planning and operation capabilities and ease the integration with third-party systems. All these demands are *Non-Functional-Requirements* because they describe how the specific issues shall be solved or improved. For a platform and framework such as provided by the developed prototype, the *Non-Functional-Requirements* are very important because their satisfaction shall ease the integration and development of modular components that provide functionality that cover concrete use-case specific tasks.

The functional requirements for an instrument system can be easily encompassed by the evaluation of existing instrument software. If the system does not come with the full function set of classical system it must at least come with mechanisms that allow to add additional functions in a fast, easy, and flexible way. This limitation would otherwise be a regression and would not constitute an acceptable solution.

3.5.1. Functional Requirements

The majority of *Functional Requirements* that apply to an instrument system can be collected from the function summary of the presented software systems in chapter 2.4. Further essential requirements for the new development have been given as constraints in section 3.3.

The following list gives a short overview over the essential *Functional Requirements.* Only the most important functionality shall be directly bound to the core system while all other functionality can be flexibly added on demand. A detailed justification for this decision and the introduction of further requirements, which are not yet clear in this abstract context, will follow in chapter 4.

The system shall ...

- ... provide a platform that allows the running of modular extensions
- ... control the life cycle of components
- ... come with functionality that allows the control of the data flow between modules
- ... provide a development API that allows access to and reuse of recurring functionality
- ... provide a User Interface that allows the configuration of the core system and its extensions
- ... provide functionality that allows monitoring of the current operation of the system
- ... allow restrictions to be placed on the user's access to critical functionality
- ... provide a layer that validates user-input in order to prevent faults caused by misuse
- ... provide a secure interface that allows access to and operation of the system remotely

Extensions shall ...

- ... perform measurements
- ... control and query hardware
- ... collect and save measurement results
- ... provide mathematical data reduction algorithms
- ... allow the creation and execution of measurement sequences
- ... provide scripting mechanisms for the creation of complex scans
- ... intelligently control the progress of the measurement using just-in-time data
- ... allow the simulation of hardware and measurement flows

3.5.2. Non-Functional Requirements

The second more comprehensive part of the requirements are the *Non-Functional-Requirements*. While the implementation of a *functional* requirement generally applies to a self-contained part of the implementation chapter 4, each non-functional requirement can be important for a larger number of independent implementation parts. A description and justification of which *non-functional* requirements have been met by each implementation detail would interrupt the readers flow and add extra redundancy.

To counter this problem, an index with all *Non-Functional-Requirements*[7] has been created in appendix 2. Each requirement comes with a *unique id*, a *description*, a *rationale* and a *fit criteria* so that a closer requirement description is not necessary in chapter 4. When a requirement relates to an implementation detail, it is specified with its unique id and links to the relevant requirement in appendix 2 starting with page 308. To ease finding a requirement they are sorted into categories and listed in the table of contents.

[7]The index also contains a minor number of functional-requirements that are referenced by multiple implementation descriptions.

CHAPTER

FOUR

OPEN INSPIRE - THE IMPLEMENTATION

Open Inspire - The Implementation covers the concrete realization of a new type of software system. Open Inspire is a minimal but well extensible prototype of a software solution that complies with all guidelines and specifications covered in chapter 3. The primary aim is not the development of a system with all imaginable features but a matured, solid and fully functional basis for many usage scenarios. Instead of implementing specific features directly in the core of the software, it is designed to be independent from specific usage scenarios due to the decoupling between the core platform and modular extensions.

The chapter starts with a short survey of the distributed *Open Inspire Architecture* und continues with a closer look at the server. An overview describes the general concept of the *OI Application Server* on page 76 before the extension system is discussed in section 4.2. With an understanding of the component architecture, section 4.3 continues with a description of the *OI Container* and the wiring of components to so called *OI Assemblies* and closes with a survey of global services provided by the server. The next section starting on page 146 covers the *OI Project Infrastructure.* It deals with the global web project used by users for collaboration and information sharing as well as as a global resource storage used by the server. Section 4.6 closes with details about the *OI Build System*, Integrated Development Environment (IDE) Integration and information for developers.

4.1. Basic Concepts and Decisions

4.1.1. Network Layout

A problem common to many scientific Data Acquisition and Control (DAC) systems is the limitation to only one communication mechanism between different elements of distributed software. Generally it is no problem when the instrument environment is designed specifically to be used by the software or when the software is developed for the particular instrument environment. However, when the solution needs to be integrateable in an existing environment [☒ 5.3.2][☒ 4.2.2][☒ 4.2.3][☒ 5.3.3][☒ 5.3.6] without changing any system parts and without violating any rules and guidelines of the operating institute or company [☒ 7.1.4][☒ 6.3.3] the problem is more difficult. To meet these requirements it is necessary to have a system that is easily integrateable in existing environments without changing them and with a minimum adaptation effort.

OI solves these issues by introducing a pluggable network architecture that is decoupled from the system core and is modularly exchangeable. This enables the reuse of existing devices with their existent protocols and network architectures and provides the ability to rapidly react to new developments without altering any part of the DAC system itself [☒ 5.3.6][☒ 5.3.8][☒ 5.3.9].

Figure 4.1 shows the network layout of a viable Open Inspire driven instrument environment. It is structured in the form of a multi-tier network with a Supervisory Control and Data Acquisition (SCADA) server in the center and an arbitrary number of Data Acquisition and Control (DAC) systems, service providers and user interface clients around. The example network is split into a secure internal area with services in the direct instrument environment and an untrusted external network such as the internet[1]. Using a firewall ❻ enables filtering network packages and restricting the transport to a selected number of services and protocols. To allow the passing of packages belonging to external clients that access the OI Server and to allow server communication with external services, OI introduces a modular network im-

[1]This is a common setup at all evaluated institutes

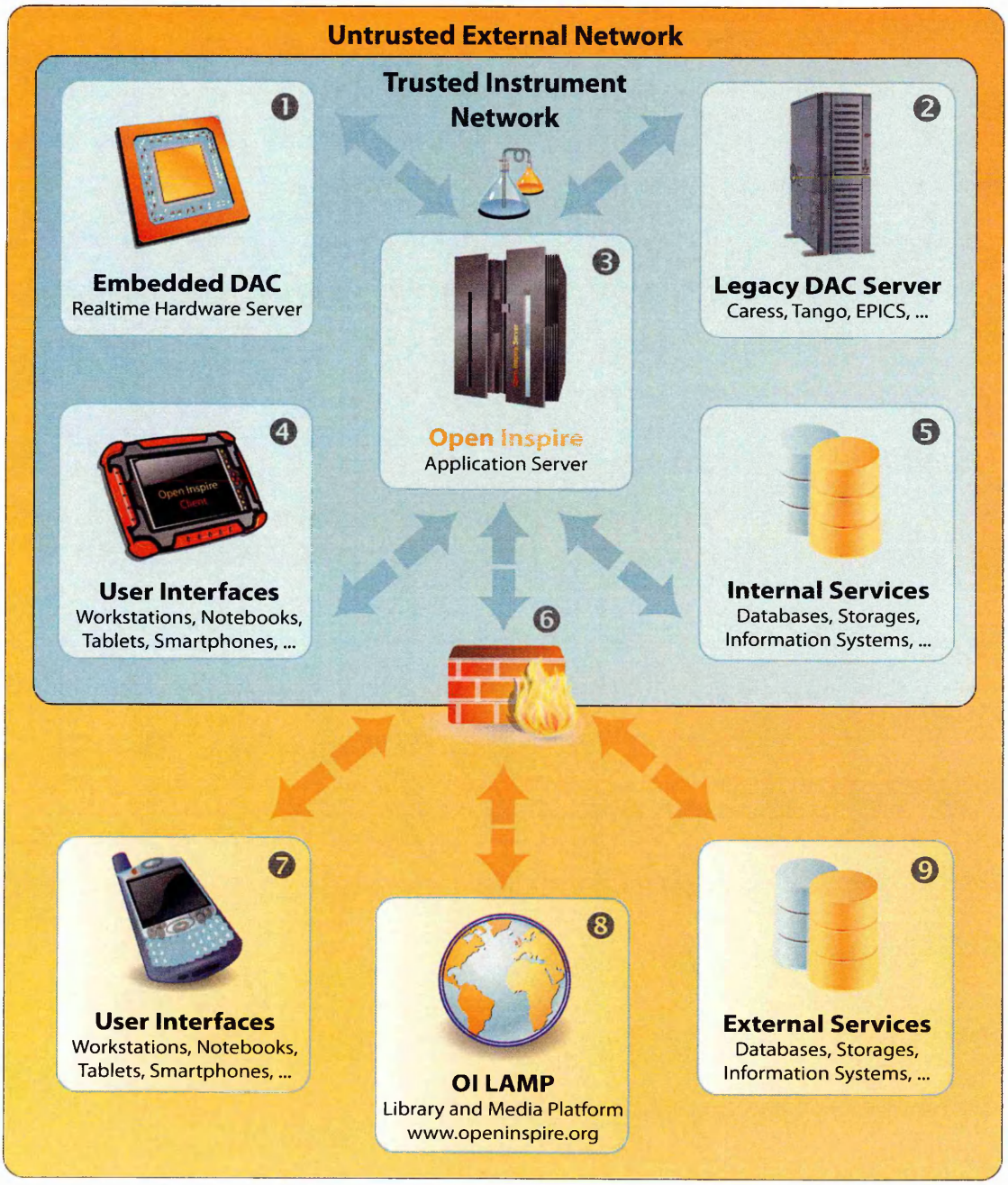


Figure 4.1.: *Open Inspire Network Layout*

plementation for a secure and encrypted communication through firewalls [☒ 5.3.5][☒ 6.5.2][☒ 6.3.2]. A firewall-friendly reference implementation is covered in section 5.1.1 on page 176.

Open Inspire Application Server

The OI Application Server ❸ acts as SCADA server, as middleware between user and different types of hardware and as a process control manager. In contrast to classical designs, the OI Server comes without task-oriented logic, without direct hardware access and without fixed user interface. Functionality can be added on demand, depending on the scope of duties. Business logic such as scan control and data processing as well as communication with DAC systems, external services and user interface clients are provided by software components and executed inside the OI Container, covered in chapter 4.1.2.

Data Acquisition and Control Services

Instead of directly access the hardware from the OI Server, this functionality has been outsourced to so called Data Acquisition and Control (DAC) servers. These servers contain all functionality needed for the direct low-level hardware access and provide lightweight network services, which allow the control and reading of data from devices. The advantage of this approach is that DAC servers can be written in any programming language, can be operated on any hardware and can provide arbitrary high level interfaces as long as a OI Server component is able to communicate with them. Different to the OI Server, which simply calls a service and waits for an answer, the DAC servers can implement hard realtime functionality needed to meet hardware timing requirements. [☒ 3.1.2] [☒ 4.2.2] [☒ 5.3.6]

For new OI based developments it is recommended to realize DAC servers in form of embedded systems ❶ that represent the functionality of only one physical device to one network service. Beside micro-controllers, programmable hardware such as Field Programmable Gate Arrays (FPGAs) are a good choice to realize critical and failsafe functionality with high performance, massive parallelization, and small latencies. For non time-critical operations with lesser safety demands one or more services can be operated on normal PC hardware and provide access to installed devices such as the camera system covered in chapter 5.6.3 on page 247.

The OI architecture also allows the integration of existing hardware control systems ❷ such as *Caress*, *TANGO*, EPICS or SICS [☒ 4.2.3]. When such systems already provide a network interface, these interfaces are often not designed to be used standalone and require additional control and conversion code. One principle of the OI design is the ability to integrate other systems without changing them, which means that all additional conversion and communication code needs to be integrated and handled by the OI server. This will be demonstrated using the example of *Caress* in chapter 5.4.

To minimize latency times and maximize the throughput between Open Inspire and DAC servers, they are located in the same network. This guarantees fewer network hops, shorter cable lengths and no speed decrease caused by packet filters [☒ 3.1.3] [☒ 3.1.7] [☒ 3.1.8].

Information Services

In addition to DAC servers for the access to low-level hardware, a second category of services can be summarized under the term information service (❺, ❾). Examples therefore are storage solutions, databases and directory services as well as monitoring services of external systems such as a neutron or synchrotron sources. To assist internal data reduction processes[2], services can furthermore act as gateways to number crunchers, computer clusters or cloud systems.

User Interface Clients

A user interface for Open Inspire can be implemented in different ways. The first way is to add it as a service directly to the Open Inspire (OI) server such as the OI *Shell* or *SysTray* covered in chapter 4.4.2 on page 143. The other and recommended way is to implement it as a network client that remotely connects to the OI Server. A client can be located directly beside the instrument ❹ or somewhere in the internet ❼. To allow the operation of clients on different types of devices, the OI Architecture makes no restrictions about the implementation, communication protocol and programming language of clients. Whenever a client communicates using a different protocol than the reference implementation, introduced in section 4.4.1, a new service protocol can be modularly added to the OI Server. Clients can provide simple monitoring functionality, can be tailored to specific tasks such as the instru-

[2]Data reduction processes transform raw experiment data to simplified, ordered and better accessible data.

ment calibration covered in chapter 5.6 or universally used for the configuration, monitoring and control of experiments such as the client OI Sunrise that is part of section 5.1.2. They can be operated as normal programs, in forms of web sites or on mobile devices such as smartphones or tablets.

OI Library and Media Platform

The Open Inspire Library and Media Platform (OI-LAMP) ❸ is a central repository for OI related libraries and media. It offers human readable content on the webpage `http://www.openinspire.org` such as manuals, tutorials and screencasts as well as machine readable services used by the OI Server, OI Clients and external tools. Open Inspire is the first scientific SCADA architecture that comes with a central component repository similar to *Apple's App Store* or *Google Play*. Components can be published by users, browsed in the web or by OI Clients and automatically downloaded and installed by the OI Server. Different to applications, OI components are not executed standalone but wired to experiments and executed inside the OI Server. More details about the OI-LAMP are covered in chapter 4.5 on page 146.

4.1.2. Application Server Basics

Compared with the systems analysed in section 2.4 the conspicuous difference of OI is that all application logic has been outsourced to a central and dedicated application server. In contrast to two-tier client-server systems, the introduction of a third tier as middleware between Hardware and User Interface (UI) prevents the application logic being interwoven with the DAC or User Interface code. Classical coupling of the application logic with the client or server code enables fast and unconfined data transfers within the software boundaries but constraints the logic to the software platform of a specific DAC server or User Interface. Using a fine grained distributed system in contrast meets the demands on a high degree of separation, reusability and scalability but requires a higher communication effort for serialization, deserialization and synchronization. This leads to higher latencies and limited throughput.

The Application Server approach of OI combines the advantages of both the systems. It offers a flexible and loosely coupled component model without compromising the good performance

of monolithic systems. Time-critical communication can only occur between the OI Server and DAC servers. But this is unproblematic, because real-time tasks are not performed on the OI server side but on the DAC servers that already reduce the data before transportation and therefore ease the timing requirements. Even less problematic is the communication between the OI server and UI clients. The entire workflow is managed by the OI server, so that a complete connection loss will not affect the server operation [☒ 3.5.4][☒ 4.1.2]. Beside these external communication, the internal data flow is nearly latency free and guarantees high throughputs [☒ 3.1.7][☒ 3.1.8]. This is mainly interesting for flexible setups with mathematical components such as measurement data reduction.

4.1.3. Application Server Internals

Considering the OI server as a *White Box* shows that it only appears to behave like a monolithic application. Internally it conforms to all claimed requirements such as modularity, exchangeability and reusability [☒ 3.7.2][☒ 3.7.5][☒ 3.8.3] . This situation is attained by virtue of to the decoupling of the server code from instrument and experiment specific functionality.

While dynamic extensions to DAC systems are usually realised by plugins, within OI they are realised in a way that is geared to application containers for web applications. Conventional plugins require a fixed interface that enables the communication between plugin and main program. This forces the main program and the plugin to follow a fixed standard concerning the interface and data structures. Outsourcing of functionality into plugins leads to a more lightweight application core, a better decoupling and enables on-demand loading but it does not solve the problem that the plugins still depend on a fixed interface and server, which is responsible for the initialization and data transport. This conflicts with the requirement [☒ 5.3.10], which demands modules to be independently usable in programs outside the OI server.

To meet this requirement, an architecture is needed that does not only make the components independent from each other but also keeps them completely decoupled from a superordinate software system. This is the job of the OI *Application Container* that uses the Inversion of Control (IoC)-pattern, which allows the movement of hardcoded references between parts

of a program to a superordinate instance to be later linked using the Dependency Injection (DI)-pattern [Fow04]. Details are covered in section 4.3.1 on page 110 and section 4.3.4 on page 128.

The OI Server hosts the OI Container and is responsible for the management and manipulation of components. In a manner similar to programs within Operating Systems, modules can be loaded into the container, configured and set to perform tasks or interact with hardware or users. Each module is independent from the container and directly useable in other unrelated software systems [☒ 5.3.10]. In contrast to programs in Operating Systems the focus does not concentrate on the execution of individual programs but on the linking of independent components to a network with an arbitrary number of building blocks. Program fragments are therefore not coupled to one specific scenario but are reusable to build up composites for many different and independent usage scenarios.

A simple material science example, would be a measurement in which a detector makes snapshots of a sample's diffraction pattern as it is rotated in steps of 5 degrees. This scenario can be built up by four modules. One for the positioner control, one to read from the detector, one to coordinate the measurement and one for the data storage. While such functionality is often directly merged with the DAC system core, the component architecture of OI does not limit the logic to one or few scenarios. Modules can be independently reused in future assemblies inside or outside OI [☒ 5.1.4]. A description of OI's *Assemblies* and of the linking of modules follows in section 4.3.2.

Figure 4.2 shows the internal schematic of the OI Server. The communication paths between external servers and clients show that these do not really communicate with the OI Server itself but with modules that are executed inside the container. The server itself is not responsible for any instrument or experiment specific work that could constrain it to a fixed usage scenario. Beyond this, a further step is to outsource all the functionality from the OI Server to the container that is not required for the essential operation. The container itself is thus lightweight [☒ 5.1.2][☒ 3.8.4][☒ 4.3.4][☒ 5.1.3] and only responsible for the installation, management and execution of modules. Tasks for specific usage scenarios such as neutron diffraction are exclusively provided by modules. The server is therefore not fixed to a specific domain

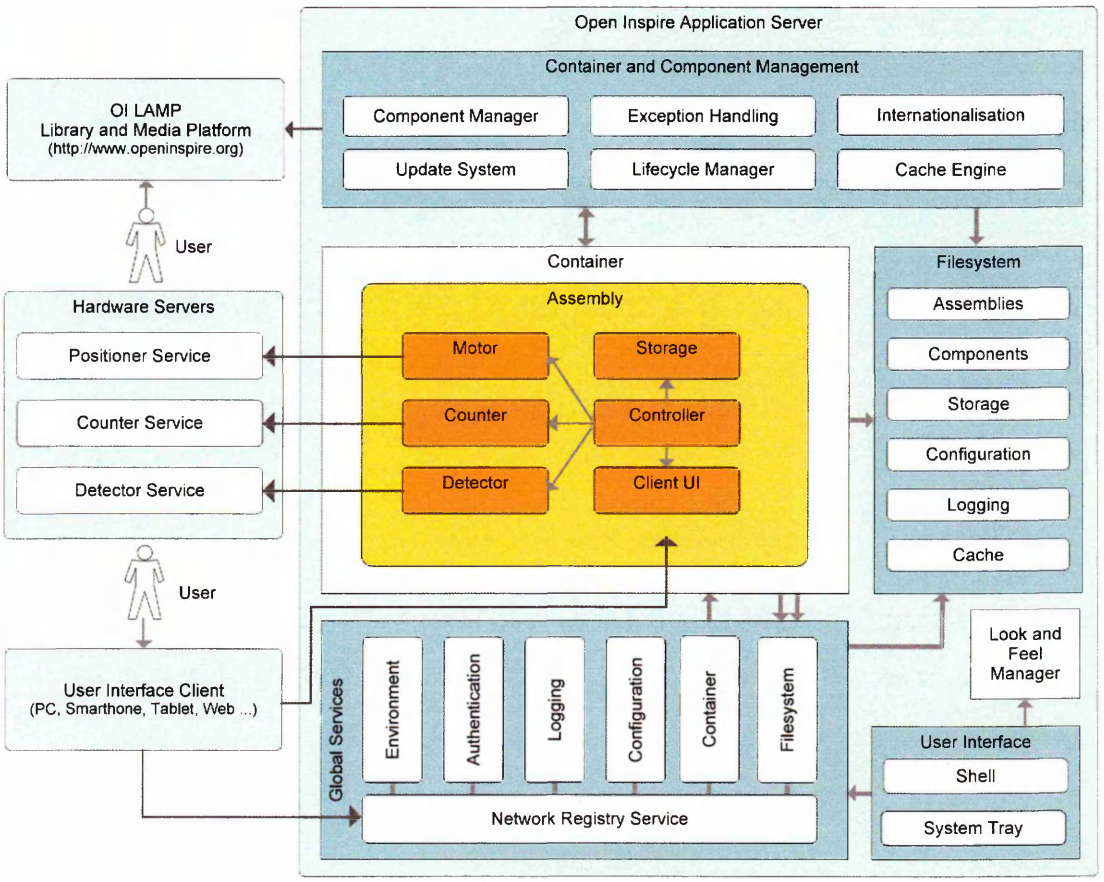


Figure 4.2.: *OI Application Server Internal View*

but can be reused for completely different use-cases such as found in *Smart Home* or industrial environments. This thinning of server functionality and the corresponding codebase, reduces the need for later modifications significantly. Changes only occur inside the container so that the framework can be kept untouched. This does not only meet the requirements for longevity[☒ 3.8.2][☒ 3.8.3] but also improves stability, error-proneness [☒ 3.2.3][☒ 3.2.5] and leads to faster understanding of the design [☒ 2.3.5][☒ 2.3.3][☒ 2.3.2].

Although most functionality can be relocated to the container, graphic 4.2 shows that some management logic of the server need to be operated outside the container. These services are subdivided into *Container Management* Services, *User Interface* Services and *Global Services*. The *Container Management* provides core functionality for the OI server and coordinates the complete lifecycle, error handling and component management. *User Interface* Services can

be plugged into the OI Server and provide user access to the server, directly on server-side. This will be covered in chapter 4.4.2. Finally the global services can be registered to provide functionality that can be used by modules to access server functionality or the filesystem. One important service is the *Network* Service that automatically provides network access to all other registered services. Clients can therefore connect to the server, authenticate by the *Authentication*-Service and access log files, start and stop the container, configure assemblies of modules or read configuration and measurement data from the filesystem. To meet all security requirements, the access to the filesystem or services can be restricted based on the registered users role [☒ 2.2.2][☒ 2.2.3][☒ 2.2.4][☒ 2.2.5][☒ 3.2.8][☒ 6.1.2].

4.1.4. Platform and Implementation Decisions

OI is platform independent and implemented using the programming language *Java*. Common languages have been evaluated and a selection made that include *C; C++*, *C#*, *Objective C (ObjC)*, *Python* and *Groovy*. The following paragraph shows the primary selection criteria and comparison between Java and the evaluated language selection.

Feasibility:

All required functional and non-functional requirements can be achieved with Java. *C* does not come into consideration due to the lack of *Object Oriented* capabilities.

Popularity and Acceptance:

According to the "Programming Language Popularity" statistics from `langpop.com` the languages *Java*, *C* and *C++* rank at the top of the normalized language popularity, acceptance and resources comparison. These results include statistics from the *Yahoo API* search, *Craigslist* database with information about job offers for programmers, *Powell's Books* with statistics about available programming books, *Freshmeat* and *Google Code* with information about languages in *Open Source* projects, *Del.icio.us* with bookmarked language topics and committed *Open Source* code at *Oloh*. A second comparison about languages people are talking about includes *Lambda - The Ultimate*, *programming.reddit.com*, *Slashdot* and *IRC*. This places *Java* on the fourth place of 34 behind *C++*, *C* and *Python*. *Python* is a good

alternative to *Java* and has been accepted as neutron experiment standard script language at the NOBUGS[3] conference 2008. Python code is executable inside the Java VM using *Jython*.

Compatibility and Platform Independence:

Java, *Python* and *Groovy* programs are directly executable on different platforms. *C*, *C++* and *Objective C* code need to be recompiled for all targets, which leads to compatibility problems when platform specific functionality is used. *Objective C* is primarily focused on *Mac OS X* and *C#* on *Windows* so that other platforms are not supported in the same way. Direct hardware access from *C*, *C++*, *ObjC* and *Python* binds the code to a platform while Java comes with an abstraction layer that prevents the loss of platform independency. With the JSR 223 : Scription for the Java Platform [jsc], Java allows the execution of currently more than 25 scripting languages including *Python*, *Groovy*, *Ruby* and *Scala* directly inside the *Java* VM. This allows the creation of OI modules with scripting code in other languages.

Error-Proneness:

Java is a strict language that prevents many faults already on code base. It will not compile until all exceptions are caught and is less prone to memory leaks since it does not allow pointer based memory access. In contrast to *Python* and *Groovy* it strictly uses hard typing, which reduces type conversion errors while programming and improves the testability. The *Java Security Manager* is important for executing OI modules inside *sandboxes*, which do not allow the access to the filesystem or restricted resources.

Language Scope and Libraries:

Java has a sophisticated language scope and comes with one of the most comprehensive class libraries with solutions for nearly all problems. Additional functionality can be accessed by libraries for the scripting languages, supported by the JSR 223.

Development Tools:

Integrated Development Environments (IDEs) to advance development and improve the time-to-market are available for all evaluated languages. For Java the most common tools *Net-*

[3]The New Opportunities for Better User Group Software (NOBUGS) conference is regular conference with the aim to improve the collaboration between IT professionals at X-Ray and Neutron sources [nob].

Beans, *IntelliJ* and *Eclipse* provide all important methods such as refactoring, autocompletion, tests, debugging, syntax highlighting and syntax checking. Section 4.6.4 on page 169 covers an example IDE integration using *NetBeans*.

Speed and Ease of Learning:

In contrast to *Groovy* and *Python*, writing small *Java* programs takes more time, caused by the verbose syntax. But this syntax has the advantage that the user is forced to write clean code that is often more reliable and helps beginners to avoid mistakes. Automated memory management and renouncing from pointers eases the development and prevents faults.

Execution Speed and Resource Consumption:

While *C* and *C++* and *ObjC* are famous for their high execution speed, the languages *Java*, *C#* and the scripting languages *Python* and *Groovy* are often considered as slow. For Java this label is antiquated. Due to language improvements over the last 10 years, *Java* comes with an improved JIT-Compiler and code optimization that make the speed differences between *Java* and *C* or *C++* minimal. In particular comprehensive programs profit from the enhanced memory management and just in time compilation and are in several situations even faster [JN]. The memory consumption of *Java* is higher due to dynamic memory management and caching mechanisms but for modern hardware this is not a limiting factor.

Hardware Access:

Except for *Java* and *Groovy* all languages allow the direct access of hardware. Java and Groovy are operated in virtual machines and designed to directly run on any hardware so that a direct access of hardware will break platform independency. Since OI is designed to be a SCADA server without direct hardware control, the platform independent optimization is more important than the access of hardware from the OI server.

OI runs on all major operating systems such as *Windows*, *Linux*, *Mac OS* and *Solaris*.

4.1.5. Survey to the Files and Folder Structure

Checking out[4] the OI Server from the *Mercurial Repository*[5] `http://hg.openinspire.org` creates a 30MB folder structure with ~800 files that will grow to ~1600 files (220MB) after building. The content and size of the directories can vary because the OI build system is able to decide, which content[6] needs to be downloaded from the Open Inspire Library and Media Platform (OI-LAMP)[7] during the build process. This reduces the required download bandwidth and minimizes the local and repository file sizes, which would be occupied by irrelevant packages otherwise [☒ 4.3.4][☒ 4.3.5][☒ 4.3.6].

The Main Folder

After building OI, the root folder *main* contains a *readme*-file and the subfolders system, mods and release. It is recommended to read the file *README.TXT* at first because it contains links and information about the OI version, building instructions and known problems.

```
main
├─README.TXT ..... Information where to start reading
├─system ..... Sources , build system and docs
├─mods ..... Sources of modules
└─release ..... Ready to run binaries
```

Figure 4.3.: *Open Inspire Root Directory*

The Release Folder

The folder *release* contains the complete and executable OI distribution. It will be created during the first build process and contains all files that are included in the OI installation packages that are downloadable from the OI website. Figure 4.4 on page 84 shows the shortened content of the *release* directory with short descriptions to each entry.

[4]The checkout process triggers the download of the sources and will be covered in chapter 4.6.1 on page 156.
[5]A *Mercurial Repository* is the storage place of sources using the Version Control System *Mercurial*
[6]The build system for instance only downloads the files necessary for the specific platform and configuration.
[7]The Open Inspire Library and Media Platform is covered in chapter 4.5 on page 146

```
main
└─release
  ├─start.sh ..... Start script for Linux, UNIX, Solaris and Mac OSX
  ├─start.bat ..... Start script for Windows systems
  ├─OpenInspire.jar ..... The Open Inspire Kernel
  ├─assemblies ..... OI Assembly Files
  │ └─Inspire 2009 ..... Demo Assemblies
  │   ├─Demo-Scan.xml ..... Scan Demo Assembly
  │   └─Demo-UI.xml ..... User Interface Demo Assembly
  ├─conf ..... Configuration Files
  │ ├─inspire.properties ..... Main Configuration
  │ └─privileges.xml ..... User and Group Permissions
  ├─cache ..... Cache for Runtime Data
  │ ├─cache.properties ..... Cache Strategy Configuration
  │ └─jni-cache ..... Cached Native Libraries
  ├─logs ..... Rotated Logfiles
  │ ├─inspire.log ..... The active Log File
  │ └─inspire-2012-09-18.log ..... A rotated Logfile Backup
  ├─oims ..... Installed OI Modules
  ├─oils ..... OI Libraries
  │ ├─unpacked ..... Installed OI Libraries
  │ └─packed ..... Downloaded OI Library Packages
  └─storage ..... Data stored by OI Modules
```

Figure 4.4.: *Open Inspire Release Directory*

The System Folder

The *system* directory is relevant for developers and contains all resources that are required to build Open Inspire (OI). This includes the sources for the kernel and libraries, all build scripts, build resources, templates, configuration files, documentation and IDE extensions. The *system* folder is covered in more detail when the build system layout is introduced in chapter 4.6.2 and illustrated in figure 4.40 on page 159.

The Modules Folder

The folder *mods* contains the sources and resources of all official Open Inspire Modules (OIMs). To achieve a clear separation from the kernel sources and libraries, these files have been outsourced to a folder that is independent from the main build system. Having the folder on the same directory level as the *system* and *release* folder also allows the easy checkout of a subset of modules[8]. A list of all modules can be found on the official website.

4.2. Open Inspire Component Design

Open Inspire comes with a system of loosely coupled components that can be used to assemble complex scenarios inside the OI container.

Java already provides a fine grained object oriented module concept, organized in forms of classes, interfaces and packages. To meet the requirements on the OI module system, this hierarchical organization of classes in packages is not sufficient. Additional functionality such as versioning, dependency handling, information hiding and the definition of public interfaces on component base are still missing. These are necessary to distribute, organize and connect components without knowing module internals or writing any line of programming code [⊠ 2.1.5].

One common modularization approach in web environments are Enterprise JavaBeans (EJBs) that come with the Java Enterprise Edition (JEE). EJBs, defined according the JSR 220, are network-compatible, provide persistence and transaction support as well as a manageable lifecycle and can be executed in any EJB-compliant container.

[8]This feature is provided my *Mercurial* Subrepositories [sub11]

One of the first OI designs was build on the EJB 2.1 standard but the development has not been pursued due to the high complexity and overhead. These drawbacks prevent occasional programmers without competent JEE knowledge, from writing modules and complicates the solution of simple problems.

Diverse independent projects try to approach the problem with component models such as the OSGi service platform [osg, All03, GPZ05], NetBeans Modules [Bou02] or the Eclipse Plugin Architecture [Bol03]. An official Java standard has been recommended as JSR 294 : Improved Modularity Support in the Java Programming Language [jsra] but never been realized. Recent developments are entitled under the name Project Jigsaw [jig] that however won't be part of the Java SE 7 [jdk] and might not be supported by other Java SE 7 implementations.

The fact that no established standard is available and that none of the approaches support a method to enable the definition of component inputs and outputs that can be easily interlinked, necessitates a new design. The primary aim here is to keep a simple and lightweight design with a high degree of reusability. One way is to introduce a wrapper language that describes the interfaces and module functionality and enables the conversion of the modules to other component models such as OSGi to guarantee a high compatibility. But defining another proprietary interface description language requires a new language to be learnt and tools that handle the language to be written.

In- and outputs of Open Inspire components are realized in forms of annotations directly inside the source code. A wrapping language is unnecessary and a programmer can come back to the language elements known from Java. The idea is not to have a description language and create a source code skeleton that can be filled with logic but to write logic and simply mark method as in- or output ports. These can be processed by the OI container or converted to interface descriptions used by other approaches or to access the modules from other languages. This design approach is flexible enough to meet all stated requirements and only requires a minimal intrusion into the source code and programming workflow.

The following sections will cover the two types of extension modules Open Inspire Modules (OIMs) and Open Inspire Librarys (OILs) that act as building blocks for complex scenarios and as wrapper for the encapsulation of libraries.

4.2.1. Open Inspire Modules - OIMs

The most important OI extension type is the Open Inspire Module (OIM). Just by the use of OIMs it is possible to create complete instrument software setups. This works even outside of the OI environment due to their strict decoupling and independence from the OI Server.

In established systems it is a common practice to have a fixed number of component types that are each customized for a specific task. The Grand Unified Model for Control and Analysis Systems (GUM)[9] [GH] for example defines building blocks such as *device drivers*, *network protocols*, *databases*, *sequencers*, *analysis* functionality or *data visualization*. One reason for the manifold number of different module types is the wish for easy to use interfaces that describe the module functionality in an intuitive way. Another reason are core designs that require modules with different types of interfaces that provide functionality for specific parts of the software. Disadvantages of the approach are the raised complexity and learning effort for the various module types as well as the need for core modifications when new types of modules need to be added. A different approach is the definition of one fixed interface for all modules[10] that needs to be generic enough to represent the functionality of all modules. This concept is flexible but error-prone because of the additional encoding and decoding effort for the transferred data and the absence of a type-safe data verification on the code level.

The OI Design makes no differences between module types. Each module follows the same design, independent from its features. Once a programmer knows the general setup of an OIM, this is valid for all other modules. The selection of a specific module type per scenario is unnecessary. Thanks to the lightweight design, a module designer can so concentrate on the logic implementation without loosing time for the creation of complex wrappers for simple module functionality.

The Users's Point of View

A user is typically not interested in the internal operation of a module but rather how it is used [☒ 2.4.7]. For this reason it is meaningful to hide irrelevant information and avoid

[9] See section 2.4.1 on page 37 for details to GumTree and the GUM
[10] Examples are the *Caress Abstract Device absdev* or the *device* interface of *Taco*

damage caused by misuse and to prevent overtax. The easier the usage of the module the higher is the acceptance.

Module Installation and Deinstallation

To use a module it needs to be installed. In most situations the installation and deinstallation of software is the responsibility of a system administrator. Installing OIMs on the contrary is trivial and thanks to mechanisms that prevent misuse promptly performable by users [☒ 5.1.5].

The manual module installation is not more than a simple copy of an OIM in the *oim*-installation folder of the OI-Server. The deinstallation is in reverse the deletion of the module file. The OI Server is able to detect changes and automatically performs all installation steps in the background [☒ 3.4.6][☒ 3.8.4]. If an internet connection is available it is yet easier. Open Inspire automatically detects when missing modules are requested, starts a download from the OI-LAMP module repository and installs them. Details to the OI-LAMP follow in chapter 4.5.

The Module's Ports and Properties

Installed OIMs can be interlinked to so called OI Assemblies that are executable inside the OI Container. While this chapter gives an introduction to the outer view on an OIM the design of assemblies and interlinking will be covered in chapter 4.3.2.

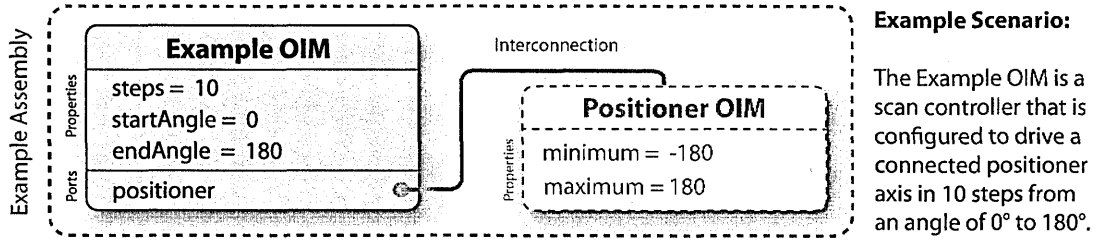


Figure 4.5.: *Open Inspire Module Example*

Figure 4.5 shows an example of an assembly that contains two modules. The focus in this section is on the *Example OIM*, which is a simple scanner module that drives a positioner

axis and can be configured with a start angle, end angle and a number of discrete positioning steps. The second *Positioner OIM* is a proxy for a concrete positioner hardware that can be driven in a range from -180° to 180°.

OIMs can be configured by two types of user configurable mechanisms:

1. *OIProperties* for the configuration of module parameters
2. *OIPorts* for the interconnection of modules

Each module can be configured by an arbitrary number of *OIProperties* such as initialization parameters, positioner limits, detector channels or storage paths. The *OIProperties* for the *Example OIM* are the start angle, the end angle and the number of positioning steps.

The second construct *OI Port* is used whenever modules need to be interconnected. The *Example OIM* provides an *OIPort* named *positioner*. This port enables the loose coupling of modules from type *positioner* such as the *Positioner OIM* with the *Example OIM*. Due to the fact that the *Example OIM* is independent from a specific positioner implementation it is possible to switch the positioner without changing universal control logic. Each OIM can have an arbitrary number of ports. Details about the internals follow in section 4.2.1.

The Developer's Point of View

From the developer's point of view it is not only important to know how a module is used but how it is build-up. The next sections will, for this reason, discuss the internals as well as the development of OIMs.

Naming Conventions

The design of an OIM requires the compliance with intentionally few conventions. From the outside it is a binary file with a unique package name that consists of three logical subsections `id_version{_suffix{#}}.oim`[11] similar to the *Gentoo Linux* [Arn05] naming for *ebuilds* [ebu].

[11] Brackets delineate optional fields and do not appear in the name. # represents any non-zero positive integer.

A distinctive name always starts with an id that contains the URI[12] of the vendor, followed by a specific module name. To improve the sortability of OIMs, all name fragments are written in reverse order, lowercase and separated by hyphens. A module named `example` with the vendor URI `http://sub.domain.tld` will under this terms have the id `tld-domain-sub-example`.

The second part is the version and reflects the development stage of the OIM. It is made up by two, three or more period separated numbers for the major and minor version followed by an optional suffix. This suffix indicates the package stage in the Software Release Lifecycle [VdHW03, Hil08, rel] and can be one from table 4.1 followed by a positive integer number.

Order	Suffix	Description
A	_alpha	Alpha release. Feature incomplete and unstable testing package.
B	_beta	Beta Release. Feature complete but still unstable.
C	_rc	Release candidate. Potentially the final product. Only bugfixes.
D		Normal release. Ready and stable software. (No suffix)
E	_p	Patch level. Bug fixes have been applied.

Table 4.1.: *Release Lifecycle Module Suffix*

Now it is possible to model a valid package name for the *Example OIM*:

`org-openinspire-oim-example_1.3.0_rc1.oim`

Adhering to these rules allows comparing OIM versions and switching to newer versions. Whenever the suffix changes[13], a package will be automatically replaced by the most recent version. Packages with different main versions such as `..._1.3.0.oim` and `..._1.3.1.oim` will in contrast coexist, because they may provide different interfaces or modified functionality.

The OIM Archive

To keep all data together, the content of an OIM is packed as a *ZIP*-archive [zip] with the suffix *.oim* that indicates its special internal file structure. Publishing an OIM as one file simplifies the distribution and circumvents problems with platform dependent file permissions, text-

[12] Uniform Resource Indicators (URIs) (*RFC 3986* [BLFM05]) are used to uniquely identify files in the WWW.
[13] For example `package_1.3.0_alpha3.oim` to `package_1.3.0_beta1.oim`

and filename encodings. All common operating systems come with *unzip* tools that also work for OIMs. Graphic 4.6 shows the complete content of the *Example OIM.*

```
org-openinspire-oim-example-1.3.0_rc1.oim
├─META-INF
│  └─MANIFEST.MF ..... Module Meta Information
└─org/openinspire/oim/example
   ├─ExampleOIM.class ..... The module logic
   └─example.png ..... The module icon
```

Figure 4.6.: *Sample OIM Archive Contents*

The example archive contains only three files. The first file *MANIFEST.MF* is located in the *META-INF*-folder and contains *meta*-data that will be covered in the next section. File two is the compiled *Java* program in the form of a *class* and file three a *png*-image as module icon. A module can contain as many *class* files as necessary as long as they stored in a subfolder of a folder structure that reflects the OIM-name[14].

The assembly and meta data of OIMs might be familiar to *Java* programmers because it is widely consistent with the Java Archive (JAR) specification [Jar]. JARs act as container for *Java Classes* and *resources* and just as well contain the mentioned *MANIFEST.MF.* Beside the high recognition value, specializations of the JAR format have been proven in different component systems such as NetBeans Modules (NBMs), OSGis, Open Document Texts (ODTs), Web Application Archives (WARs) or Enterprise Application Archives (EARs). The OIMs specification only extends the *MANIFEST.MF* with additional meta entries that do not clash with the original JAR-specification. Changing the filename suffix from *.oim* to *.jar* transforms an OIM to an unconfined Java Archive that can be used in any independent *Java* application.

[14]When the module id is `org-openinspire-oim-example`, the classes must be located in a subfolder of `org/openinspire/oim/example`

The Meta Description

The extensions of the OIM manifest compared to the standard JAR manifest will be exemplified with help of the *Example OIM* manifest that is illustrated in figure 4.7.

```
1  Manifest-Version: 1.0
2  Ant-Version: Apache Ant 1.7.1
3  Created-By: 14.3-b01-101 (Apple Inc.)
4  Built-By: Stefan Alexander Flemming <sf@openinspire.org>
5
6  OIM-Name: org-openinspire-oim-example
7  OIM-Version: 1.3.0-rc1
8  OIM-Title: OIM - Example Module
9  OIM-Info: A Module that moves an axis in steps between two angles
10 OIM-Icon: org/openinspire/oim/example/example.png
11 OIM-Authors: Stefan A. Flemming <sf@openinspire.org>
12 OIM-Weblink:
13   http://oim.openinspire.org/org-openinspire-oim-example
14 OIM-Service: org.openinspire.oim.example.ExampleModule
15 OIM-Mod-Deps:
16   org-openinspire-oim-domain-science_1.3.0-rc1
17 OIM-Lib-Deps: oil-oi_1.3.0-rc1
```

Figure 4.7.: *OIM Manifest Example for the Example Module*

A JAR manifest contains a list of "name:value"-pairs[15] inspired by the RFC822 standard [rfc].

The first four lines describe the version of the manifest format, the version of the build system that has been used to create the manifest, the version and vendor of the Java implementation and the name and email address of the person that created the JAR file.

Beside these *JAR* keywords, it is allowed to introduce an arbitrary number of new keywords without compromising the compatibility. Open Inspire uses this feature beginning with line 6 to introduce 10 new keywords that are all prefixed with *OIM* to avoid clashes.

OIM-Name contains the unique name of the OIM in the same way as it is used for the filename and **OIM-Version** the appropriate version id. These entries allow to identify or repair the OIM filename even when it has been renamed.

OIM-Title contains a human readable name and **OIM-Info** a short description of the OIM. Both values are used by tools for the management, installation and documentation.

[15] A description of all keywords, optional sections and comments can be look up in the JAR specification [Jar].

OIM-Icon declares a path to a graphic that can be used by tools to show an icon that represents the OIM. Scaling pixel graphics is afflicted with quality loss so that multiple graphics can be optionally stored with different resolutions[16].

The field **OIM-Authors** contains a comma separated list of developers including their email addresses in squared brackets.

OIM-Weblink is an URI to continuative information about the OIM. Official OIMs link to a related information page on `http://www.openinspire.org`.

OIM-Service specifies the path to the service class that publishes the public interfaces of the OIM. More information about service classes follow in section 4.2.1 on page 95.

OIM-Mod-Deps and **OIM-Lib-Deps** finally define dependencies to other modules OIMs and libraries OILs. The fields can be either left blank or can contain one or more colon separated entries with complete OIM or OIL names including their versions.

Main focus of attention in conjunction with *loose coupling* is to reduce the dependency factor between modules so far that changes will have a delimited and controllable effect on the system. OIMs that need to communicate with other modules therefore have to define a dependency to the module type of the counterpart in the module manifest. The OI Container will otherwise effectively prohibit any hidden communication between OIMs and blocks all mutual access until a dependency has been explicitly defined [☒ 6.2.5].

Whenever a module is loaded by the OI container, the container automatically determines and installs all required modules. Even mixing the same module type with different versions is possible. Outside the container, when an OIM is treated as a JAR, all dependencies need to be resolved manually and communication restrictions between modules can not take effect. Beyond that, mixing of different OIMs will fail through class path clashes.

[16] To support images with different resolutions, a suffix such as `_16.png` for a 16x16 pixel image can be added. Allowed suffixes are `_16`, `_32` and `_64` in combination with the graphic format `.png`, `.gif` or `.jpg`.

The Developer's View of the Module Interface

The development emphasis for the OIM interface model was on a design that is characterized by a high flexibility, simple to learn and easy to integrate into existing applications [☒ 2.3.5][☒ 2.3.6][☒ 2.3.7]. Looking back to the manifold interface designs of the software systems in chapter 2.4 makes clear that the diverging demands make it difficult to find an universal interface approach [☒ 3.1.8][☒ 3.7.4]. On page 87 it has already been mentioned that the interface design can be roughly separated into a generic and a customized approach. Generic interfaces are more flexible, not limited to specific module tasks and do not run the risk of being incompatible with prospective module functionality while customized interfaces are more intuitive, easy to use and fail-safe.

The most commonly practiced approach is the definition of one generic module interface for the communication between all modules and a superordinate framework. For non object-oriented systems this interface often consists of several fixed functions such as *init*, *cleanup*, *read* and *write*. All modules can be registered in the same way at a superordinate instance which is able to initialize them with *init*, finalize them with *cleanup* and transfer data in both directions using *read* and *write*. In object-oriented systems, it is customary to define a base class or interface that contains equivalent methods for *init*, *cleanup*, *read* and *write* and let new modules inherit these methods. The inheritance from a common *super class*, for example, allows putting all derivates in a polymorphic list and iterate this list to call *init* an all modules. Similar models can be found in the *Abstract Device* of *Caress*, the *TACO/TANGO* device, the *Device*-class of *Frack* [Fle05] or even in the modules of the *Linux Kernel* [LO05].

The definition of a generic interface with fixed methods and fixed parameter types however does not solve the problem of how to transfer module specific data. One option is to pass data packages with arbitrary content between the *read* and *write* methods. This can be both copies and references to an explicitly defined data structure that is interpretable by any module. Another way, which is used by distributed systems in particular, is the utilization of streams. Local streaming implementations can, for example, be implemented by the *Java Serialization*

API or *Pipes*. A common design pattern is the *Dataflow Pattern*[17] [dat, MK03], which is heavily used in *LabView* to realize the communication between components. For distributed instrument systems CORBA and character streams, such as *XML* streams, have been widely accepted[18].

Both versions require each module itself to take care about the stream serialization, deserialization and data interpretation. Implementing these functions individually for each module causes a high redundancy and consequently increases the development effort, maintainability effort and error-proneness. Providing this functionality in contrast by external libraries or the framework binds the modules to external resources and violates their self reliance. All solutions have in common that the user is confronted with an additional level of complexity, which constricts the maintainability and learnability [☒ 2.4.7][☒ 2.4.8]. The serialization and deserialization furthermore have a negative impact on the data rate.

To give occasional programmers without expert knowledge the opportunity to quickly understand the module's interface concepts and promptly write modules, the OIM design follows a new less interfering path. Instead of defining new interface definitions and data transfer guidelines, modules will be interconnected using already existing and well known constructs of programming languages. Within object-oriented systems it is common to interlink classes by references. The first class therefore defines a method with a parameter that expects a reference to an instance of the second class. This mechanism is self-explanatory, familiar to all programmers and failsafe, because the compiler is able to check the validity of method names, parameters and the matching of referenced instances with the parameter type.

The question arising at this point is how to interlink modules from the outside. Up to now it is only possible to define references between objects directly in the source code so that they are compiled to be part of the immutable binaries. Here one of the most important design decisions comes in. Thanks to the OI Container it is possible to move references between modules from the source code to a superordinate layer and let the container inject them.

[17] The *Dataflow Pattern* is the counterpart to the control flow model in text-oriented programming languages and used in graphical programming languages to describe the information flow between graphical logic blocks.

[18] CORBA is for instance used by *Caress* and *TANGO*, XML streams by *Frack* and DANSE.

To enable this injection, the service class[19] of the OIM needs to publish information about the methods that should be treated as externally accessible ports. Several base structures for service classes have been evaluated[20] but only Plain Old Java Objects (POJOs)[21] turned out to meet all required qualities. POJOs are lightweight, do not follow any major object models or complex conventions, are familiar to all *Java* programmers and independent from external resources or frameworks. This makes them predestinated for loosely coupled object models.

> *"We [Fowler, Parsons, MacKenzie] wondered why people were so against using regular objects in their systems and concluded that it was because simple objects lacked a fancy name. So we gave them one, and it's caught on very nicely."* [Fow00]

Figure 4.8 illustrates the source code for a valid OIM service class, which can be used for the *ExampleOIM* in figure 4.6 on page 91. The ports *startAngle* and *endAngle* in figure 4.5 have been removed for clearness and internally replaced by the fixed values 0° and 180°.

On first view, the source code does not differ from any other *Java* Class except for **line 15 and 20**. The Class *ExampleOIM* declares two private attributes *omegaAxis* and *steps*, which are encapsulated by public Setter- and *Getter*-methods. The *Setter* at **line 16** allows setting the *omega*-axis of the *ExampleOIM* by passing an instance of the type *Positioner*. A second Setter- and Getter at **Line 21 and 25** are used for the number of steps in which the *omega*-positioner should drive a semicircle.

The method *performMovement* at **line 29** is responsible for the axis-movement. It iterates with the preset number of steps from an angle of 0° to 180°, initializes the external positioner with the target angle for each step (**line 31**) and starts the positioning (**line 32**).

Even now it is possible to use the *ExampleOIM* in any ordinary *Java* application by passing a *Positioner* instance to *setPositioner*, setting the number of steps with *setSteps* and starting the axis movement with *performMovement*.

[19] The service class has been defined in the OIM manifest under the key *OIM-Service*
[20] e.g. EJBs, *Spring Beans*, MBeans, CORBA stubs and skeletons, ICE Slices and XML
[21] POJO has been mentioned the first time in 2000 by Martin Fowler and is since 2005 a common term used to distinguish lightweight object models from models with many dependencies and complex conventions.

```
1  /*************************************************************
2   *          Sourcecode for a simple axis movement            *
3   *************************************************************/
4
5  package org.openinspire.oim.example.ExampleOIM;
6  import org.openinspire.lib.*;
7  import org.openinspire.oim.example.Positioner;
8
9  public class ExampleOIM{
10
11     private Positioner omegaAxis;
12     private int steps;
13
14
15     @OIPort
16     public void setPositioner(Positioner omegaAxis){
17         this.omegaAxis=omegaAxis;
18     }
19
20     @OIProperty
21     public void setSteps(int steps){
22         this.steps=steps;
23     }
24
25     public int getSteps(){
26         return this.steps;
27     }
28
29     public void performMovement(){
30         for(int angle=0;angle<180;angle+=180/steps)
31             omegaAxis.setSetpoint(angle);
32             omegaAxis.move();
33             // wait for axis / perform operation
34         }
35     }
36 }
```

Figure 4.8.: *Source Code for the Service Class of the Example OIM*

Using it inside the *OI* Container is no problem either but is at this stage only meaningful when all methods are publicly accessible by the container. This is disadvantageous because internal operations would be unprotected and it moreover leads to an unclear, uncontrollable and error-prone design. Distinguishing between *public* and *private* methods will not help either because *public* methods are required for the communication between classes inside modules. This does not automatically mean that accessing the module from the outside is safe, too.

The solution for this problem is the specific selection of all methods that should be treated as ports. One way is the declaration of them inside a further description file. The advantage of this approach is that any class can be equipped with meta information without changing the class itself. Drawbacks are the extra effort for the maintenance and synchronization of the meta file with the source code as well as the overhead, additional learn expenses and the fixation to a syntax that needs to be kept compatible with new releases. Early versions of the *OIM* model made use of such XML based meta files but these have been replaced by the upcoming *Java Annotation* design in 2006. The JSR 175 [jsrb] specified *Annotations* as a mechanism in *Java 1.5* that can be used to directly place meta data in the source code.

OI introduces the two *Annotation* types *@OIPort* and *@OIProperty* to simplify the *OIM* interface and get rid off external meta data. Whenever a method should be marked as a port, it is sufficient to add *@OIPort* prior to the method name. This is similar to the *@OIProperty*, which can be added to declare a method for the configuration of module parameters.

Figure 4.8 illustrates the use of *Annotations* in **line 15 and 20**. The first one declares the method *setPositioner* as an *OI Port* and allows connecting any external *Positioner* module to the *ExampleOIM*. The second one defines *setSteps* as *OI Property* to make the number of positioning steps configurable from outside the module.

Line 9 shows that an *OIM Service Class* does not need to extend a superclass or implement an interface. This is an important design decision, because languages which do not support multiple inheritance would otherwise disallow using existing classes as *OIM Service Class* when they are already included in an inheritance hierarchy.

A second important step that makes OIMs independent is the abandonment of mandatory library or OIM dependencies. The import statement in **line 7** exclusively imports the *Positioner* interface since it is explicitly used by the *Module Example.*

One small disfigurement is certainly inevitable since it is not possible to use *Java Annotations* until they are previously defined. One option is to define the *@OIPort* and *@OIProperty* annotation by including a code snipped (figure 4.9) into the module. A second option is to reduce the redundancy but add a dependency to a module that provides the required annotations as in line *line 6* of figure 4.8.

```
1  package org.openinspire.lib.meta;
2  import java.lang.annotation.*;
3
4  @Documented
5  @Retention(RetentionPolicy.RUNTIME)
6  @Target(ElementType.METHOD)
7  public @interface OIPort {
8      String description() default "No description";
9      boolean mandatory() default false;
10 }
11
12 @Documented
13 @Retention(RetentionPolicy.RUNTIME)
14 @Target(ElementType.METHOD)
15 public @interface OIProperty {
16     String description() default "No description";
17     boolean mandatory() default false;
18 }
```

Figure 4.9.: *@OIProperties and @OIPort Annotations*

Generating Modules

While the last sections covered the details of the OIM design, a developer will normally use a generator to simplify the module creation. A generator asks for the name, version and module properties such as authors and dependencies, validates the information and creates an OIM with the correct name, folder structure and files (figure 4.6). This folder includes a manifest (figure 4.7), an icon and a service class template (figure 4.8), which allows adding ports and properties by adding the appropriate annotations prior to the method names.

4.2.2. Open Inspire Libraries - OILs

A second newly introduced module type is the Open Inspire Library (OIL). In contrast to OIMs, this module type is not used to provide mechanisms that allow interconnecting modules or set properties but encapsulates already existing binary libraries. Since OILs have a unique id and version similar to OIMs they can be set as module dependency and automatically resolved and installed by the container. In contrast to OIMs, OILs are not limited to be used by other modules but also by the container and build system to access external functionality. Benefits are the central installation and update management as well as the decrease of redundancy.

While the next section shows how to wrap platform independent Java libraries using OIMs, the subsequent section shows how OILs can be used to wrap and include even platform dependent libraries, written in other languages.

Platform Independent Java OILs

Binary *Java* libraries are published by means of JAR files which can contain compiled classes, media such as graphics, audio files, videos or other documents. To enable OI to use the large number of existing JAR files, OI wraps these files into OILs to ease the use and solve several problems.

In classical designs, JARs need to be loaded at program start and make all their functionality globally and unrestrictedly available to the whole application. In OI it is however not clear at the program start, which libraries are required by OIMs that are subsequently loaded by the container.

Since installing all JARs prophylactically that are ever used makes no sense, including all JARs that are required by a module into it is the only option. This leads admittedly to a high redundancy since the same JARs are again and again bundled with different modules. The solution for this problem is to define a JAR as dependency, which is installed when the first OIM requests the library and will be afterwards shared with other OIMs. The problem

is, that this is not possible with classical JARs due to the lack of a naming and versioning mechanism that enables the identification and referencing of JARs in a distinct namespace.

This problem is solved by means of OILs that encapsulate existing JARs without changing them but by adding a unique id and meta information. This mechanism will be made clear by the OIL `oil-xml-1.3.0-rc1-all.oil` that provides XML creation and parsing functionality.

The OIL Naming Scheme

An OIL is published similar to an OIM but this time with the file ending *.oil*. This ending is always prefixed with *-all* when it is platform independent, which is the case for JARs. The further name elements embrace the name and the version of the OIL. Since JARs can not be assigned to a general namespace, they can be named more flexibly than OIMs. The name must however be unambiguous, begin with *oil-* and mirror the contents of the OIL. The version is used for the OIL and not for the enclosed JAR file and follows the already known OIM convention, which has been introduced on page 89. The OI Container is therefore able to load even different versions of a library at the same time.

The OIL Archive Content

Figure 4.10 illustrates the contents of the example OIL `oil-xml-1.3.0-rc1-all.oil`.

```
oil-xml_1.3.0-rc1_all.oil
├─ oil.properties ..... Meta Information for the OIL archive
├─ jars ..... Folder with wrapped JAR libraries
│  └─ jdom.jar ..... Example for a wrapped library
└─ licenses ..... Folder with licenses for the libraries
   └─ APACHE_LICENSE.TXT ..... Example License for wrapped library
```

Figure 4.10.: *Platform Independent OIL Archive Contents*

The archive contains a file *oil.properties*, which comprises *meta* information and the two folders *jars* and *licenses*. All JARs are encapsulated in their unaltered form, how they are published by the developer, and located in the folder JARs. The folder *licenses* contains

license files that need to be included for legal reasons and confirmed by the user during installation [⊠ 7.1.3].

The Meta Description File

The unique filename of an OIL is already sufficient to make the server recognize and manage an OIL but does not yet show the function of the OIL. This is done by the included file **oil.properties**, which provides additional meta information that can be queried by management tools or when creating a list of all OIMs and their description on a webpage. Figure 4.11 shows the contents of the meta file for the example OIL.

```
1  ######################################################
2  #     Open Inspire Library Wrapper Archive V1.0      #
3  ######################################################
4
5  # The unique id of the library wrapper archive
6  oil.wrapper.id=xml
7
8  # The version of the library wrapper archive
9  oil.wrapper.version=1.3.0-rc1
10
11 # A human readable name of the library wrapper
12 oil.wrapper.name=The JDOM XML Library Wrapper
13
14 # A human readable description for the wrapper
15 oil.wrapper.info=\
16     Library wrapper for the JDOM Java XML Library.
17
18 # The build date of the library wrapper
19 oil.wrapper.date=20101124
20
21 # The version of the wrapped library
22 oil.jar.jdom.version=1.1.1
```

Figure 4.11.: *Example of a Platform Independent OIL Properties File*

Beside comments, this file contains several *key-value*-pairs. **oil.wrapper.id** and **oil.wrapper.version** contain the name and version of the library to repair the filename when it has been renamed by mistake. **oil.wrapper.name** and **oil.wrapper.info** contain a human readable name and description of the OIL. **oil.wrapper.date** is used for the creation date of the OIL and **oil.jdom.version** contains the versions of all encapsulated JARs. To allow assigning

versions to more than one included JAR it is important to add the name of the JAR as the third period-separated substring.

The user is in normal situations not confronted with this meta-file. Developers can utilize an OIL generator that encapsulates the libraries, license files and generates the *meta*-file similar to the OIM generator.

Native Libraries

Using existing JARs from Open Inspire allows access to the considerable functionality provided by third-party suppliers. Enlarging this repertoire to include functionality written in other languages will, however, again increase the number of available libraries. This is inevitable when the functions are either not available or portable[22] to *Java*. An example is the NeXus API, which is written in *C* but which is required for the storage of measurement data in the NeXus format, which is covered in section 2.4.4 on page 49 and 5.2 on page 190.

Java already provides mechanisms to access native libraries written in *C / C++* by means of the Java Native Interface (JNI). Using this API however is accompanied by several disadvantages such as the loss of platform independence due to the fact that the native libraries are compiled for solely one OS and processor architecture [☒ 4.3.3][☒ 4.3.6][☒ 5.3.3]. A second problem is that all libraries need to be globally installed in the executing OS and registered in the library path before a *Java* program that requires them can be started.

The lack of mechanisms that allow the installation of native library during runtime however leads to conflict because it is unknown to the OI Server, which libraries will eventually be requested by OIMs that are executed in the container. Prompting the user to install required libraries manually restarting the OI server is not an option, because it leads to a high complexity, maintenance effort and is not accordable with the dynamic component model of OI.

The solution for this problem is to extend the OI Container by a mechanism that manipulates *Java's* internal library management to allow loading subsequently installed native libraries

[22] Reasons are for instance restrictions in commercial libraries, missing source codes, high complexity or a disproportional porting effort.

from arbitrary sources. To approach the problems related to the lack of platform independency, two encapsulation mechanisms for native libraries have been considered.

One option is the integration of native libraries for all platforms in the same OIL. The advantage is the simple maintainability because only one file is be distributed, independent from its architecture. The drawback is that one OIL contains libraries for all platforms, which leads to a larger file. Downloading and installing this file results in a higher consumption of bandwidth and storage space[23]. For this reason, Open Inspire uses a second variant that splits the native libraries to multiple OILs that each represent one platform [☒ 4.3.3]. This approach is more elaborate but consumes considerably less resources.

Naming Conventions

For platform independent OILs such as the `oil-xml-1.3.0-rc1-all.oil` it has already been quoted that the filename contains the keyword *"all"*, which stands for *"all architectures"*. For platform dependent OILs, this keyword will be replaced by a designator that indicated the architecture and Operating System, required to load the native library. Valid designators are *windows*, *linux*, *macosx* and *solaris* combined with the architectures *x86* or *x64* for 32 or 64 bit architectures[24]. A valid filename for the platform dependent NeXus OIL for *32 Bit Linux* can for example be `oil-nexus-1.3.0-beta2-all.oil`.

File Contents

The only difference in the file structure of a platform dependent OIL compared to a platform independent is an additional *jnis*-folder. This folder contains all native libraries for the platform that has been specified by the filename. For *Linux* and Solaris the libraries are stored as *Shared Objects (.so)*, for *Mac OSX* as *Dynamic Libraries (.dylib)* and for *Windows* as *Dynamic Link Libraries (.dll)*.

Example 4.12 shows the content of the NeXus OIL for *32 Bit Linux*. In addition to the native libraries in the *jnis*-folder and the NeXus-license in the *licenses*-folder, this file contains an

[23] A native library with 10 MB would result in a 90MB OIL when libraries for 10 architectures are included.
[24] This list may be extended by PPC, MIPS or ARM architectures.

additional *jnexus.jar*. This file is provide interfaces that allow accessing the native code in a *Java*-friendly way.

```
oil-nexus_1.3.0-beta2_linux-x86
├── oil.properties
├── jars
│   └── jnexus.jar
├── jnis
│   └── libhdf5.so.0, libhdf5.so.0.0.0, libjnexus.so, libjnexus.so.0,
│       libmxml.so.1, libmxml.so.1.0, libNeXus.so, libNeXus.so.0,
│       libsz.so.2, libsz.so.2.0.0
└── licenses
    └── COPYING
```

Figure 4.12.: *Native OIL Archive Contents*

The Meta Description File

As shown in figure 4.13, the meta file **oil.properties** of a platform independent and platform dependent OIL differs in only two additional entries.

The entry **oil.jni.qualifiers** contains a list with names of all loadable native libraries in the OIL. This is important when the internal name of the native library differs from the filename because *Java* identifies a library by its internal name and not filename.

oil.jni.version finally contains the version of the encapsulated native library. Not to be confused with the version of the OIL, which is set by **oil.wrapper.version**.

It is important to keep in mind that native OILs are always workarounds and will never replace well designed and platform independent *Java* libraries.

4.2.3. Component Management

Using the loosely coupleable and easy maintainable component model improves the handling of extensive DAC environments. It systematically allows the hiding of complexity and the

```
1  ######################################################
2  #     Open Inspire Library Wrapper Archive V1.0      #
3  ######################################################
4
5  # The unique id of the library wrapper
6  oil.wrapper.id=nexus
7
8  # The version of the library wrapper
9  oil.wrapper.version=1.3.0-beta2
10
11 # A human readable name of the library wrapper
12 oil.wrapper.name=The Native Nexus Library  Wrapper
13
14 # A human readable description of the library wrapper
15 oil.wrapper.info=\
16     The Wrapper for the native NeXus API\
17     to store measurement data in hdf4, hdf5 and xml.
18
19 # The creation date of the library wrapper
20 oil.wrapper.date=20090707
21
22 # A comma separated list of native library qualifiers
23 oil.jni.qualifiers=NeXus
24
25 # The original version of the wrapped native library
26 oil.jni.version=NeXus-4.2.0 MacOSX
```

Figure 4.13.: *Example of a Platform Dependent OIL Properties File*

automation of recurrent tasks so that users and programmers are only confronted with simple and manageable interfaces. The design not only saves time that otherwise would need to be invested in learning and understanding complex unbounded software structures, but also paves the way for further improvements such as easily acquiring an overview of existing modules, the finding of the correct module for a problem and retrieving and keeping it up to date. Here it is not sufficient to simply install and run a module, but to identify, resolve and download all module- and library dependencies for all modules in a recursive way. This expenditure of time can be reduced by using a central component repository and tools for the automated management of modules and libraries.

Component Repository

The OI Component Repository is a central store for official OIMs and OILs and part of the Open Inspire Library and Media Platform (OI-LAMP), covered in section 4.5 on page

146. All components are globally accessible via `http://store.openinspire.org` and can be downloaded in a manual or automated manner [☒ 4.4.2][☒ 4.4.3][☒ 4.4.4][☒ 4.4.5][☒ 3.8.4][☒ 3.7.5]. In contrast to decentral solutions, it ensures that modules with older versions are kept downloadable and referencable by assemblies even after years [☒ 3.8.2][☒ 3.8.3].

The server is not limited to a fixed repository but allows the entering of arbitrary URLs in the server configuration. This enables the setting up of local repositories for filtered network environments[25], thereby decreasing the network load or operation of overlays with modified component compositions, which is useful when non-public modules are to be made accessible or global modules blocked. To advance the reliability and pursue load sharing, more than one randomly picked repository URL can be set in the configuration[26].

Usage of the Repository

To alleviate the repository usage, the *oi-client* library provides an API for searching, downloading and updating components and retrieving module meta information from the server.

The **OI Website** at `http://www.openinspire.org` uses this API to generate a browseable web index of all components and their corresponding meta information. It allows users to get an overview, comparing and finding modules without downloading them. New components can be uploaded are published after manually testing their validity, safety and reliability.

A second consumer of the API is the **OI Component Browser**, which comes as an installable application. It allows browsing the component database, directly installing OIMs and OILs including their dependencies and can generate, verify and validate new modules.

The **OI Server** itself uses the API for its whole component management, dependency calculation, update service and for reading meta information.

Definite Referenceability

Another important aim for a global repository is the absolute and unambiguous referenceability of components, which at the same time form the basis for further applications:

[25] The HZB for instance does not allow accessing the internet from the instruments.
[26] Per default the URLs `http://store1.openinspire.org` to `http://store9.openinspire.org` link to different mirror servers

One example are scientific papers. Adding an OI Assembly to a publication would allow the exact reproduction of the software setup for the experiment, to be repeated at any later time. To keep publications small, the OI-LAMP furthermore supports the central web publishment of *Assemblies* and inserting a simple link to a paper instead of the whole content.

Another aim is to provide a mechanism to enable the finding and correcting of failures retrospectively [☒ 6.2.5]. For example, when an *Assembly* is added to the measurement data and it turns out that the setup, or a module, was faulty some situations allow the measurement data to be corrected or declared invalid.

4.3. Container and Assemblies

With Open Inspire Modules, section 4.2 introduced a component design that is configurable using OI Properties and interconnectable via OI Ports. While using these modules as standalone components is possible, the real benefit of the design becomes clear when they are managed by an application container. This section starts with a design consideration, the OI Assembly Design and closes with a survey to the OI Container.

4.3.1. Design Consideration

Classical Design Approaches

Using and interconnecting OIMs independently from the OI Server can be done by instantiating, configuring and linking OI Service Objects directly in the source code. A program therefor needs to add all required OIMs and dependent OILs to the *classpath*, import the *Service Classes*, configure, instantiate and start them in the correct order[27]. Properties need to be configured manually by calling *OIProperty* annotated setters with configuration values and calling *OIPort* methods with references to other *Service Class* instances.

This procedure is cumbersome and leads to a close coupling between modules and main program. It is inflexible, difficult to maintain and time consuming, since all module parameters and links between modules may be modified exclusively by programmers with source code access.

[27] Instantiating, configuring and starting OIMs in the wrong order causes a crash due to a missing dependencies.

One possible improvement is the introduction of a user interface for the module configuration. Parameters for a setup are so variably customizable for specific environments but the composition and linking of modules remains static and only changeable by programmers on source code level.

A different approach particularly suitable for experiments that need to be changeable at short notice are configuration files such as the *hardware_modules.dat* of *Caress* (B.3.1) or the *config.xml* of *Frack* that allow the specification of the loadable modules and their parameters. Using this approach a parser searches the configuration file for known devices and their configuration parameters and plugs them into the system. This system however assumes that these devices are known by the core system before they can be configured or used for measurements. Setups with new and unknown devices are still not usable without changes to the core system to add support for the new device.

A more flexible approach is component wiring and configuration by scripts. Such scripts could be executed by the OI Server, which in this case will be responsible to control the access to installed modules. The advantages of this approach are:

- From within a script it is possible to instantiate[28], interconnect and configure an arbitrary number of modules without limitation for a small set of configuration parameters but the full strength of the selected scripting language.
- Using one script per instrument setup allows to quickly switch between different coexisting scripts for different types of experiments.
- Modules with so far unknown interfaces are usable, because the developer itself takes care about the correct data interpretation instead of an automated system.

The drawback of the approach is that a script author can hardly be forced to comply with all good practices and programming conventions. A script that is erroneous or irreconcilable with the overall design can not only influence the system stability and security but moreover lead to the following problems or constraints:

[28]Instantiation is the creation of an instance of an object in an Object Oriented Programming language.

- The host system can be harmed or compromised by bypassing security mechanisms [☒ 6.1.2].
- The exchangeability of scripts between instruments can not be ensured, because using of platform dependent or incompatible ressources from within scripts can not be prohibited [☒ 5.3.3].
- The unconfined complexity of scripts prevents a machine based or graphical creation and modification of scripts.
- Programming skills are required for the creation and modification of scripts [☒ 5.1.4].
- Extra time and effort is inevitably required for learning the scripting language, scripting conventions and good practices [☒ 2.3.3].

Solving these problems by mechanisms for the automatic verification and validation of scripts is either only partially realizable or not at all. Beyond that it is impracticable and too extensive due to the large scope of common script languages.

Container-based Design

The insights gained by the analysis and review of instrument systems as well as the considerations in the last section shows a core problem of classical systems: The tight coupling of experiment setups with the base system. This is inter alia a reason why nearly all scientific disciplines operate their own instrument software that covers only a specific scope of experiments. The concomitant redundancy is particularly obstructive since base functionality is reinvented over and over, instead of extending existing systems with specific functionalities.

One solution to these problems is the consequent separation of all experiment specific software parts from a base system, which is limited to provide a core functionality for various independent application scenarios. This state can be reached by abstracting the base functionality so far that it is no more related to a specific application domain. Changing an application specific part of the software system will therefore consequently have no influence on the base system. The advantage of this approach over a tailored system for material scientific scenarios is the availability of an immutable basis for various application scenarios. Users and

developers can so work on one software core that is sharable by scientific disciplines such as astronomy or even industrial process or smart living scenarios [☒ 5.3.10][☒ 5.3.9] [LCBM05].

To achieve such a system it is essential to reach higher decoupling levels than realized by commonly used instrument systems. A necessary first step is the complete decoupling of the system base from modular extensions as has already been realized by the OI component design described in Section 4.2. The next step is the separation of experiment specific setups from the system base. This includes not only the parameterization of the OI modules but also the experiment assembly with individual OIMs. To achieve this, all statically compiled references between modules need to be moved to higher management instance that is in the position to create, configure and interconnect OIM-instances.

In Open Inspire the OI Container has been designed specifically for this task. It is able to create flexible networks of OI modules, configure them and handle their life cycles. The core design of the container builds on a combination of the Inversion of Control (IoC) and DI-pattern [Fow04]. Using the IoC pattern, the supervision of the control flow between modules can be extracted from the application code and relocated to a level that is dynamically maintainable by the OI Container. Calling *OIProperties* and *OIPorts* for the configuration and interconnection of modules is therefore externally controllable by the OI-IoC-Container using the Hollywood Principle: *"Don't call us, we'll call you"* [MBF99]. The interconnection of OIMs is finally realized by calling *OIPort* annotated methods with instances of interlinkable OIMs[29] using Dependency Injection.

While some instrument systems already profit indirectly from IoC-Container architectures such as provided by Enterprise JavaBean (EJB) Containers or Rich Client Platforms (RCPs)[30], these techniques have never been tailored for instrument systems. Common container architectures found in application servers are primary optimized for web environments and come with a richness of functionality that will never be used by instrument systems. The same applies to containers in Rich Client Platforms that are optimized for graphical clients applications. The added complexity directly effects the maintainability, stability and learn-

[29]Detailed information about the injection process will be covered in section 4.3.4 on page 136.
[30]TACO / TANGO → EJB) [GTP+03], GumTree → Eclipse RCP [GM]

ability of such systems because containers need an overhead to be prepared for this custom functionality.

For the early Open Inspire developments[31], the choice first fell on the lightweight IoC Container of the *Spring Framework* [JHA+]. The simple and scalable architecture is characterized by its versatility and can be found both in web and RCP-applications [HO07]. In OI it has been embedded into the system core and used for the creation of object networks using Dependency Injection with help of so called XML-based *Spring Context* files.

With the introduction of the OIM and OIL model it became clear that the adaption of *Spring* to meet the requirements of OI would take longer than developing a new tailor-made solution. This applies particularly to the integration into the build system, simplified context files for the wiring of OI modules and the OIM/OIL-lifecycle management. A further concern is the ongoing *Spring* development such that the dependence on the framework would require regular adjustments and a close contact with developers to stay up-to-date with the latest version[32].

With respect to the requirement for the reuse of existing technologies [☒ 5.3.2][☒ 5.3.6], further less complex container architectures with minimal overhead such as the *Pico* and *Nano* container [Fow04, pic11] have been analyzed. These containers are characterized by their simplicity, compactness, application neutrality and independence from other libraries. But compared to *Spring*, an extra effort is required for implementation of essential features such as XML-based configuration architecture. Considering that the time needed for the implementation of the core features provided by the evaluated containers is small against the time required to add OI specific module management functionality.

For the above reasons and the provision of a better check on the development and independence from other projects, the decision was made in favor of developing a new and tailored design. Since 2007 OI comes with its own SCADA optimized IoC-container that complies with all requirements for a flexible instrument system. While chapter 4.3.4 will cover the internal

[31] Until version OI 2007.0
[32] This includes bug fixes, replacing deprecated functionality and following new good practices.

design and functionality of the OI Container, the following chapter gives an introduction to the OI Assemblies, used for the configuration and interlinkage of OI Modules.

4.3.2. OI Assemblies

With the relocation of the module management from the statical server- and module code to the responsibility of the container, users are able to create and modify complete experiment setups in an easy and flexible way by so-called *OI Assemblies.*

An *OI Assembly* is a simple XML-based configuration file that describes the exact module configuration and interlinkage of an experiment in a simple and handy way. Due to the concentration of all information in one small file it is easy to convey and distribute.

The XML-based format makes it both human and machine readable. Since it is a text file, fast modifications can be made with any ordinary text or XML editor. To allow this in an easy and intuitive way without much training, only a small selected number of XML-tags have been defined. The requirements for an optimal machine based processing are fulfilled by the entirely interpretable and unique semantic of *OI Assemblies.* In contrast to script based solutions, *OI Assemblies* are automatically and losslessly convertible between different formats and read/writable by XML-tools that are available for all programming languages and operating systems.

Installing an *OI Assembly* requires no more than the copying of the file to the *assembly*-folder of the OI Server. Once it is executed, the OI Container will automatically determine all required OIMs/OILs, download missing ones from the OI-LAMP, configure them and create the module interconnections. A more detailed illustration of these activities will follow in chapter 4.3.4 after illustrating the general design and features of OI Assemblies by several examples.

The general Assembly Layout

Figure 4.14 shows the content of a complete and valid *OI Assembly*. To prevent problems with special characters, it is recommended to start an assembly with setting the text encoding to UTF-8 [33] and the XML-version to 1.0.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2
3  <oi-assembly
4     xmlns="http://openinspire.org/schema/assembly"
5     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
6     xsi:schemaLocation="http://openinspire.org/schema␣␣␣
          oi-assembly-1.3.xsd">
7
8     <spec>
9        <title>PhD Thesis Example Assembly</title>
10       <version>1.0</version>
11       <date>2011-03-01</date>
12       <originator>Stefan Flemming, sf@openinspire.org</originator>
13       <abstract>
14          A short Description of the OI Assembly.
15       </abstract>
16       <description>
17          A comprehensive Description of the OI Assembly.
18       </description>
19    </spec>
20
21    <oim id="positioner" type="org-openinspire-oim-phd-positioner"
          version="1.0.0-rc1">
22       <property name="minimum" value="-180.0" />
23       <property name="maximum" value="180.0" />
24    </oim>
25
26 </oi-assembly>
```

Figure 4.14.: *Layout of a minimal OI Assembly*

The relevant content of the file starts with the root tag **oi-assembly**, which can have exactly one **spec** and an arbitrary number of **oim**-tags. To allow an OI independent validation of the XML file[34], the lines **4** to **6** are used to assign the document to a unique name space.

To describe the assembly as a whole, the **spec**-tag from line **8** to **19** allows the entering of a set of global metadata. The tags **title**, **abstract** and **description** contain the name, a short and a more comprehensive description of the assembly; **version** and **date** are used

[33] The 8 Bit Unicode Transformation Format (UTF-8) is an international text encoding (RFC 3629 [Yer03])
[34] Every XML editor can so check the syntax and structure of the *OI Assembly*

to track changes and `originator` contains the name and contact of the author. These tags are primary used by tools that allow browsing or listing context files such as the OI-LAMP online browser.

Line **21** finally introduces the `oim`-tag. This tag is used to create and configure an OIM instance as well as interconnecting it with other modules. The example in figure 4.14 only shows the creation of a single OIM instance while real setups normally have various instances. When the assembly is started, the OI Container checks if the OIM `org-openinspire-oim-phd-positioner` in the version `1.0.0-rc1` is already installed. If not, a download from the OI-LAMP with a subsequent installation will be performed so that the container can create an instance of the module. This instance gets the unique name *positioner*, which can be referenced by other modules in the container. Beyond that, the two two properties *minimum* and *maximum* are set using the values *-180.0* and *180.0*.

Assembly Basics

While figure 4.14 only covers the creation of a single OIM, figure 4.15 shows a more comprehensive example that also illustrates the interconnection of modules. For clarity, all further assemblies are illustrated without metadata such as the text encoding, schema declarations and assembly specification. Although omitting these information is not recommended, the assembly keeps anyhow executable. The container will in this situation replace missing data with default values.

The example describes a setup of a simple neutron diffraction experiment composed by a *positioner*, a *detector*, a *counter* and a software based *scanner* for the experiment control. The OIMs created in line **3**, **8** and **13** are used to control concrete hardware devices. The devices are configured by *OIProperties* as already known from the *positioner* but with the difference that the *counter* uses the property *dimensions* instead of *limits* and that the *counter* has no properties at all. The *scanner* in line **16** finally coordinates the experiment sequence. It can for example use the positioner to rotate the sample from 0° to 90° in 10° steps and use the counter to wait 60 seconds at each step while the detector meaures incoming neutrons.

```
1  <oi-assembly>
2
3     <oim id="positioner1" type="org-openinspire-oim-phd-positioner"
          version="1.0.0-rc1">
4        <property name="minimum" value="-180.0" />
5        <property name="maximum" value="180.0" />
6     </oim>
7
8     <oim id="detector1" type="org-openinspire-oim-phd-detector"
          version="1.0.0-rc1">
9        <property name="xchannels" value="16" />
10       <property name="ychannels" value="16" />
11    </oim>
12
13    <oim id="counter1" type="org-openinspire-oim-phd-counter"
          version="1.0.0-rc1">
14    </oim>
15
16    <oim id="scanner" type="org-openinspire-oim-phd-scanner"
          version="1.0.0-rc1">
17       <property name="start" value="0" />
18       <property name="end" value="90" />
19       <property name="steps" value="10" />
20       <property name="period" value="60s" />
21       <port name="positioner" ref="positioner1" />
22       <port name="detector" ref="detector1" />
23       <port name="counter" ref="counter1" />
24    </oim>
25
26 </oi-assembly>
```

Figure 4.15.: *A simple OI Assembly for a Scan*

The creation and configuration of software components by configuration files is nothing new and is used in other instrument systems. However, the difference here is how Open Inspire uses these files, additionally, for the creation of dynamically configurable component networks. This is concretely realized by the *OIPorts* in the *Assembly* that have been introduced in Section 4.2.1. To give the scanner access to the *positioner*, *detector* and *counter*, these three modules are referenced using the *ref*-attribute of the *port*-tag in line **21** to **23**. It is only necessary to pass the *id* of the referenceable OIM. The identification of the respective instances and the dependency injection will be automatically performed in the background. Internally, the instances of the modules are passed to the methods *setPositioner*, *setDetector* and *setCounter* of the Scan OIM after checking if the parameter type of the *OIPort* annotated method is compatible with the referenced OIM. The order of the OIMs in the OI Assembly

has no effect since the correct creation and injection order is calculated by the OI Container according to the module dependencies.

When the scan sequence is to be changed, this can be achieved straightforwardly by customizing the scan properties in line **17** to **20**. If it is intended that the assembly is to be used for different hardware, only the hardware modules such as the *positioner*, *detector* and *counter* need to be replaced. The scan itself remains untouched.

Multiple Module References

While the *scanner*-module in Figure 4.15 only illustrates referencing one module per port, the next example describes how to connect multiple modules to the same port. The precondition for this multiple referencing is that an *OIPort* is able to accept more than one OIM instance. This condition is provided for ports with a parameter type that is a derivate of the interface *java.util.collection*[35].

A demonstration of this feature will be given with the aid of the *WidgetTestcenter* OIM, which is used to debug assemblies and modules. It can be used inside an OI Assembly like other modules but exceptionally opens a graphical windows instead of working in the background. One possible realization for such a GUI could be a statically arranged set of widgets for the visualization and control of OIMs data that is passed through a fixed set of ports. The main drawback of this approach is the low flexibility since each new widget arrangement requires its own new GUI module. The *WidgetTestcenter* solves this problem in a more flexible way by using only one port with name *components* that accepts an arbitrary number of different OIMs. Whenever the *WidgetTestcenter* is started, it automatically displays a matching widget for each connected OIM. The source for these widgets is the OIM *Widgets*, which contains a multitude of graphical elements for the control and visualization of OIMs.

Figure 4.16 shows the configuration of multiple references with help of an assembly that is part of each OI 1.3 installation. In the same way as in the previous example, the first step is the creation of the three devices *positioner*, *counter* and *detector* in line **3**, **8** and **11**. But this time these modules are to be controlled and visualized by the *WidgetTestcenter*, which

[35]The interface *java.util.collection* is part of the standard Java SDK and implemented by lists such as *LinkedList*, *ArrayList* and *Vector* and sets such as *HashSet* or *TreeSet*

```
1  <oi-assembly >
2
3      <oim id="positioner1" type="org-openinspire-oim-demo-positioner"
           version="1.3.0-m5">
4          <property name="minimum" value="-180.0" />
5          <property name="maximum" value="180.0" />
6      </oim>
7
8      <oim id="counter1" type="org-openinspire-oim-demo-counter"
           version="1.3.0-m5">
9      </oim>
10
11     <oim id="detector1" type="org-openinspire-oim-demo-detector"
           version="1.3.0-m5">
12         <property name="horizontalSize" value="256"/>
13         <property name="verticalSize" value="256"/>
14     </oim>
15
16     <oim id="widgetTestcenter"
           type="org-openinspire-oim-debug-widgettestcenter"
           version="1.3.0-m5">
17         <port name="components" ref="positioner1" />
18         <port name="components" ref="counter1" />
19         <port name="components" ref="detector1" />
20     </oim>
21
22 </oi-assembly>
```

Figure 4.16.: *Multiple Referencing Example - The Widget Testcenter*

is created in line **16**. Connecting the three modules with the *WidgetTestcenter* requires not more than calling the port *components* mutliple times with the *IDs* of the devices. The conversion to a *Collection*, required by the *OIPort components*[36], is automatically performed in the background.

Executing the OI Assembly opens the window in figure 4.17 and shows widgets for the three devices. These devices are software simulations of real hardware and so already controllable by the widgets. Only a few lines of configuration make it possible to compose an arbitrary number of module arrangements. When more than one *WidgetTestcenter* is used in an OI Assembly, a window for each one is opened which allows multi screen usage. Is a module connected with multiple *WidgetTestcenters*, the corresponding widget is shown and changed

[36]Internally, the method of the port in the *WidgetTestcenter* is realized by: `@OIPort public void setComponents(List<Object> modules) {...}`

in all windows synchronously. It is possible to freely move and resize a widget. All widget settings are saved when a window is closed and automatically restored the next time.

Regular Expression based Referencing

With *Multiple References*, a simple mechanism has been introduced that allows the connection of more than one module per port. This mechanism is very handy for small scenarios but for a large number of modules it can become cluttered by referencing each module individually.

To reduce complexity and improve productivity, the container has been extended to allow Regular Expressions (RegExs) in ports. Regular Expressions [Fri06] are a commonly accepted and powerful standard for the filtering of texts by defined patterns. In the OI *Assembly* they are applied to connect a pattern-based selection of OIMs to a port.

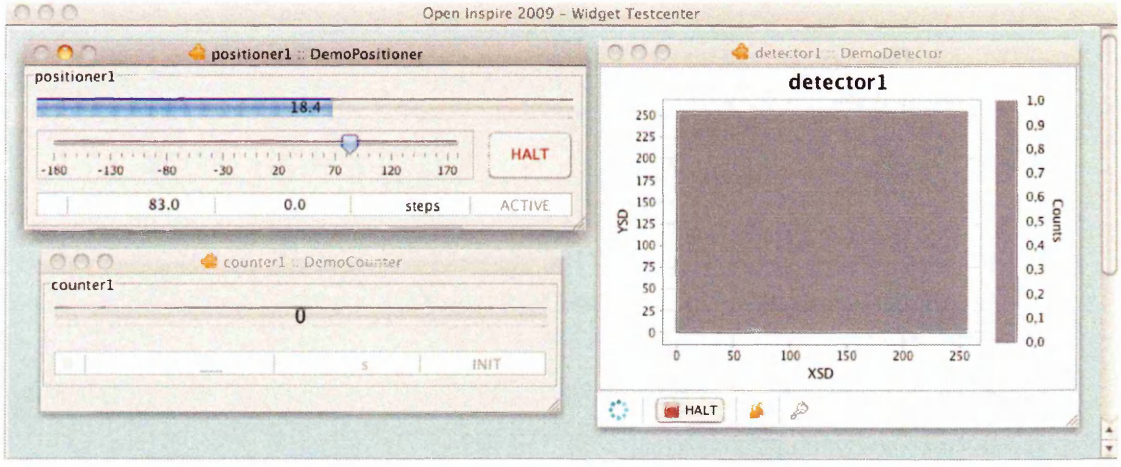


Figure 4.17.: *Screenshot of a Widget Testcenter Window*

A concrete example is line **19** of assembly 4.18. Instead of referencing each OIM individually, the wildcard '*' is used to connect all OIMs in the assembly with the port. The OIM *WidgetTestcenter*, which provides the port, is however automatically excluded from the selection to prevent recursion effects.

Table 4.2 shows a selection of simple example patterns but is not limited to this small subset. To apply complex patterns, the OI assembly allows to utilize the full strength of the filter syntax, provided by the *Java Regular Expression Engine*. Using RegEx in small assemblies

```
1  <oi-assembly>
2
3     <oim id="observerPositioner"
          type="org-openinspire-oim-demo-positioner" version="1.3.0-m5"
          role="observer">
4        <property name="minimum" value="-180.0" />
5        <property name="maximum" value="180.0" />
6     </oim>
7
8     <oim id="userPositioner"
          type="org-openinspire-oim-demo-positioner" version="1.3.0-m5"
          role="user">
9        <property name="minimum" value="0.0" />
10       <property name="maximum" value="360.0" />
11    </oim>
12
13    <oim id="adminPositioner"
          type="org-openinspire-oim-demo-positioner" version="1.3.0-m5"
          role="admin">
14       <property name="minimum" value="-200.0" />
15       <property name="maximum" value="200.0" />
16    </oim>
17
18    <oim id="widgetTestcenter"
          type="org-openinspire-oim-debug-widgettestcenter"
          version="1.3.0-m5">
19       <port name="components" ref="*" />
20    </oim>
21
22 </oi-assembly>
```

Figure 4.18.: *Using Regular Expressions and Roles in OI Assemblies*

such as example 4.18 only has a marginal effect, using patterns in comprehensive assemblies leads to a noteable simplification.

Role-based Security Mechanism

A closer look at figure 4.18 reveals that Regular Expressions are not the only new feature. Line **3**, **8** and **13** introduce a new *role*-tag that is part of Open Inspire's role-based security mechanisms [☒ 3.2.8][☒ 6.1.2]. OI uses a user database to assign each user to a specific role[37]. After authenticating by username and password, a user gains access to all services that are activated for his role. Beside global container services, which will be covered in chapter 4.4, the same user access control mechanism is also applicable for OIMs inside the container.

[37]Details about the OI Authentication Service and config will be covered in section 4.4.1 on page 140

Expression	Selected Modules
*	All modules
.*Positioner$	All modules, which end with *Positioner*
^user.*	All modules, which start with *user*
observerPositioner|userPositioner	The module observerPositioner and userPositioner

Table 4.2.: *A selection of common Regular Expression*

Adding a *role*-tag to an OIM entry means that only users that belong to the given role may access it. The three roles *observer*, *user* and *admin* are defined by default but an arbitrary number of additional roles can be freely added and populated with users[38]. Beyond that, inheritance hierarchies between users are possible. The default group *user* for example inherits all privileges from *observer* and the *admin* the privileges of all other roles. When a person who belongs to the *user*-role starts the assembly 4.18, the *WidgetTestcenter* will only show the widgets for the *userPositioner* and *observerPositioner* but due to missing privileges not for the *observerPositioner*.

Roles are an essential security feature that allows restricting the access to harmful devices to trusted users [☒ 6.1.2]. To prevent bypassing the security mechanisms, writing and manipulating assemblies is reserved for privileged users.

Graphical Representation of Assemblies

The intentional abandonment of commands, control structures and other elements that control the data-flow in OI Assemblies allows a simple, complete and lossless conversion between different description forms. A particularly important form is a graphical representation of an assembly [☒ 2.1.5][☒ 2.4.9]. In contrast to script-based configurations, it is possible to create a definite and complete representation directly from the assembly. Vice versa Open Inspire comes with a graphical editor that allows creating and changing assemblies without ever being confronted with the syntax and internals. Beside graphical assembly editing that will be covered in section 5.1.2 in more detail, another possible way is converting assemblies to formats such as HTML or LaTeX. These formats allow an optimal integration in websites and print-documentation.

[38] Adding roles and users to the server configuration is explained in Section 4.4.1 and in the configuration file

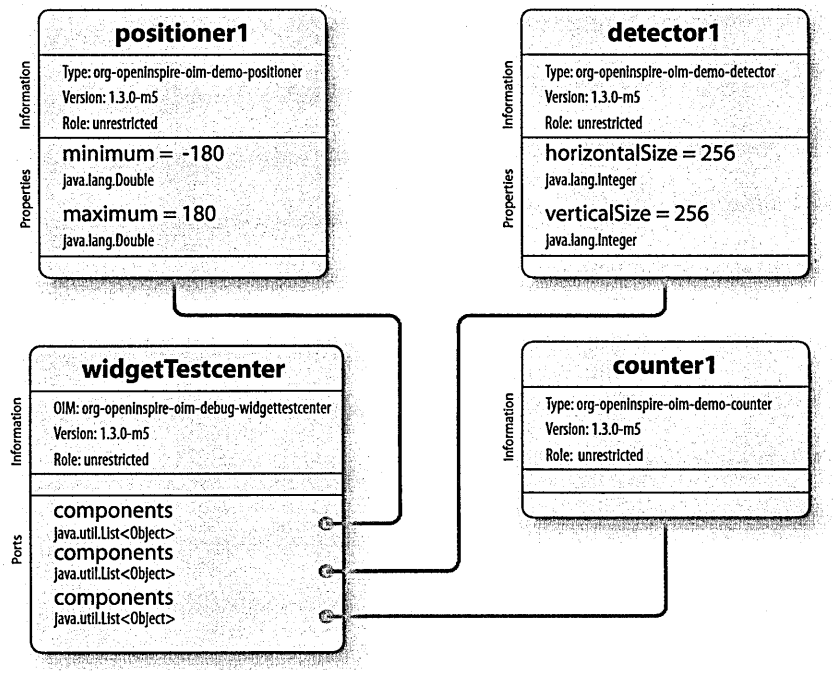


Figure 4.19.: *Graphical Representation of a simple OI Assembly*

Since assemblies can be visualized in a graphical form without information loss, all further OI Assemblies in this work will be printed as graphics instead of text excerpts. A first example is figure 4.19, that shows a graphical representation of assembly 4.16 on page 118.

Dissemination of Assemblies

An important advantage of OI Assemblies is the fact that one single file is necessary to describe and reconstruct a complete experiment setup. The inclusion of the OIM version in the assembly makes it possible to reconstruct a former setup even after years in exactly the same way.

In classical systems, changes are often performed at the core level and published as a whole under a new version. The disadvantage here is that it cannot be guaranteed that old setups will work in the same way as the time when they were created and tested.

With the OIM versioning, a fine grained versioning system has been introduced that allows creating or loading assemblies that may even contain outdated modules. The container therefore autonomously downloads the archived OIMs that were used by the assembly at the publication date. When the assembly should now be brought up to date, outdated modules can be gradually replayed by new ones. Even mixing OIMs modules with different versions is possible.

An important feature is the subsequent correctability of systematic measurement errors. If it becomes known that measurements are corrupted due to a faulty version of a module, these can be identified by the provided assembly and in many situations corrected subsequently [39].

A different area that benefits from OI Assemblies are publications. If an assembly is attached to a publication, it contains all data required to understand and reconstruct the exact experiment setup. Due to space restrictions it is however often impossible to include the entire assembly in textform. In this situation it is possible to publish the assembly similar to OIMs at the OI-LAMP and insert a unique link to it in the publication.

4.3.3. OI Interface Domain Model

The previous chapters covered the internal setup, configuration and interlinkage of OIMs. Example 4.15 showed how to restrict an OI Port by fixing the setters's parameter type to a specific module type. Assembly 4.16 and 4.18 on the contrary illustrated how to connect an

[39] OI is the first scientific system, which allows storing a textual single-file system setup beside measurement data that enables to automatically reconstruct a complete software setup.

arbitrary number of modules to a single OI Port by passing a *Collection* of references. The responsibility for the interpretation and processing of passed objects here directly delegates to the OIM.

Regarding the first option and using a concrete parameter-type for a port such as *org-openinspire-oim-demo-positioner* does not meet all requirements on flexibility, reusability and interchangeability. The scanner OIM is limited to only one *positioner*-OIM, which is specified by the fixed port type, although other *positioners* with similar interfaces and functionality could be used as well. However, if each module is accepted by the *scanner*, it is not externally verifiable, if a connected module is compatible at all.

A solution for this problem is the definition of common interfaces thereby ensuring that only devices are connected to a port that implement one of them. While defining a common interface for a few locally used device types is unproblematic, the definition of an universal interface standard raises organizational problems. Such universal standard would require a consensus on the definition of interfaces in the correspondent scientific discipline.

Time and organizational reasons make this only achievable by an expert commission and not as part of a PhD thesis.

The only way around this difficulty is the definition of a local interface standard for a bounded selection of use cases. The main challenge is now to minimize the problems in the run-up that will arise when the local standard should be later adopted to a globally accepted standard. Typically, the extension points for modularly expandable systems are directly provided and tightly coupled to the core system. Making changes to these interfaces is very problematic because it breaks the backwards compatibility and requires the adoption all modules that make use of the interface. This is the reason why interfaces will be never changed normally. Instrument systems are designed for long-term operation so that their interfaces need to be defined in a way that does not restrict its application in future scenarios. To achieve this, they typically need to be defined in a very general and abstract way. Such unspecific interfaces however have the drawback that they are less intuitive and require a higher maintainability effort and complex logic for the interpretation of interchanged data.

Interface Domain Basics

To address these problems and improve flexibility, an extended interface concept has been developed for Open Inspire. Instead of integrating the interfaces directly in the OI Server, even the interfaces have been mode modularly exchangeable. To achieve this in a way that fits optimally in the OI Design, no new mechanism has been developed but the interfaces themselves have been outsourced to OIMs to be shifted to the OI Container.

OIMs that exclusively provide interface specifications are called *OI Interface Domains.* Since these are full-fledged and valid OIMs they can be defined as dependency of other modules, published by the OI-LAMP as well as automatically installed and managed by the server. With regard to contents they only differ in the lack of program logic and *Service Classes*, because no OI Ports or OI Properties are required. This is necessary to keep open the option of converting the interfaces to other standards or harnessing them for other programming languages.

Figure 4.20 shows an excerpt of the *OI Interface Domain org-openinspire-oim-domain-science*, which provides interfaces and enumerations for the use by diffraction setups. If a module makes use of such a standard interface, it only needs to define an OIM dependency to the *Interface Domain* and make sure that the *Service Class* implements the appropriate interface. An example is module *org-openinspire-oim-demo-counter* whose *Service Class* implements the *Counter* interface from the introduced *Interface Domain.* If the parameter of a port is of type *Counter*, arbitrary OIMs can be connected that implement this standard interface. The scanner can thus be used without modification, with counters of various hardware controllers.

Interface Inheritance

OI Interface Domains directly benefit from inheritance hierarchies between interfaces. The interface *Counter* for example comes not only with the methods *startCounting()*, *stopCounting()* and *resetCounts()* but also inherits all public methods of the interface *Device.* This in turn inherits from *DeviceState*, *Emergency* and *ChangeableValue*

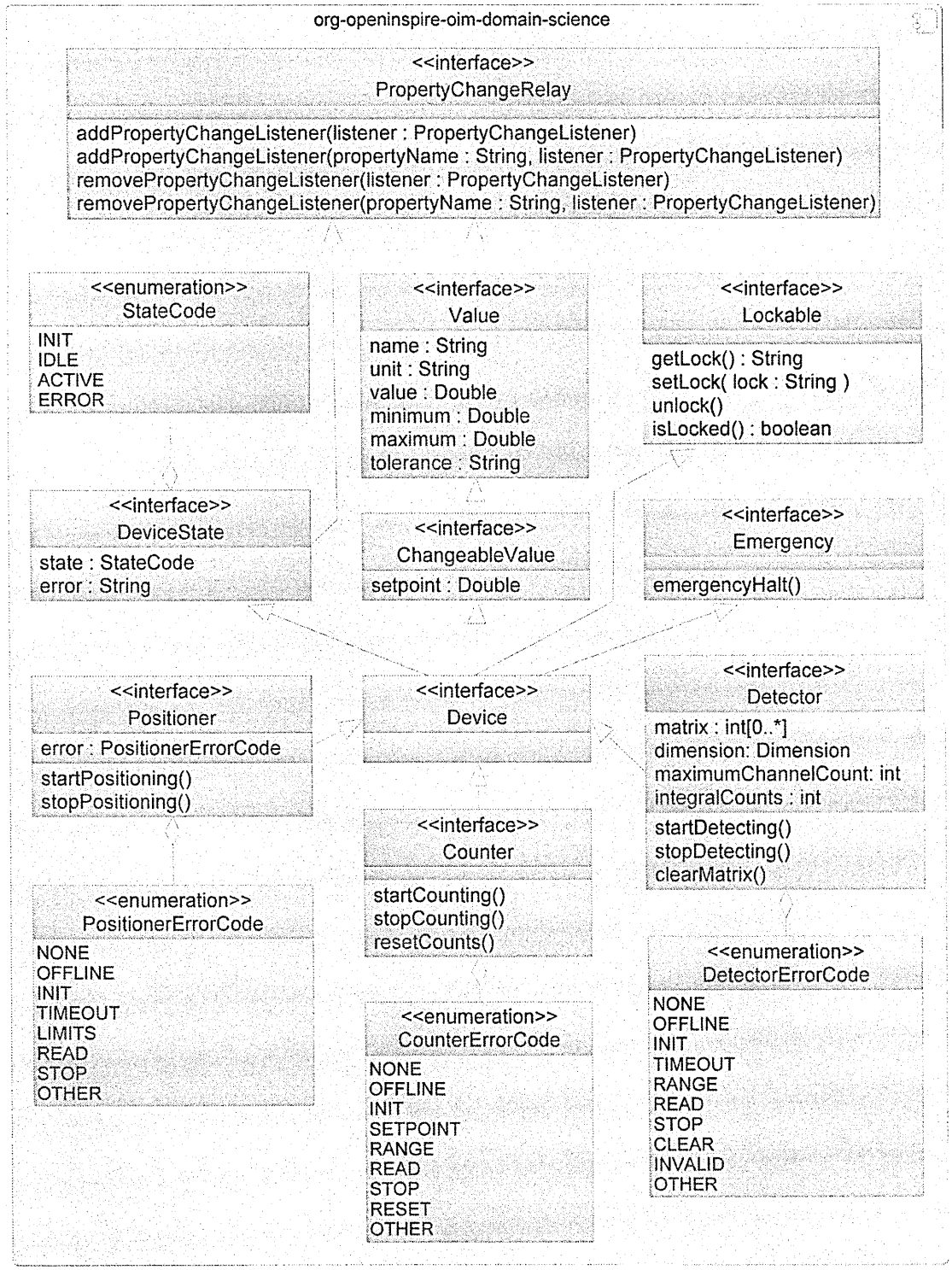


Figure 4.20.: *Excerpt from the OI Science Domain*

Example 1: An *EmergencyShutdown* module, which has a port of type *Emergency* can be used to halt all modules that are connected with it and implement the interface *Emergency*. Setting the wildcard '*' for this port in an *OI Assembly* automatically interlinks all *Counter*, *Detector*, *Positioner* and *Device* objects with the module. The *EmergencyShutdown* OIM can so iterate all connected modules and stop them by calling *emergencyHalt()*.

Example 2: A second example that makes use of interface inheritances are GUIs. The *WidgetTestcenter* for example is able to select a fitting widget for an OIM according to its implemented interface or superinterfaces. If a widget for a specific positioner is to be displayed, a first check determines if a suitable widget for the exact OIM is available. If this is false, a more general control for the *Positioner* will be used. If that is also missing, the interface hierarchy will be iterated until a widget is found for *Device*, *ChangeableValue* or *Value*.

A different case are combined graphical components such as lists, which can display cross-sectional information of different OIMs like detectors, counters and positioners in the same widget[40]. Not to be neglected are configurable GUIs that allow users to create widget compositions by selecting the best fitting presentation form of a module depending on the current use case.

Benefits of Interface Domains

The outsourcing of the interfaces to OIMs does not only have an impact on overall flexibility but is also a pre-condition for further requirements:

- If a general interface standard is to be defined at a future date, this can be quickly deployed as OIM by the OI-LAMP. Changes on the OI Server and downtimes at the instruments are not necessary. The transition to new standards is seamless.
- *OI Interface Domains* can coexist. Assemblies of modules that use older interfaces remain valid and usable. Required *Interface Domains* will be installed automatically.

[40] All three modules implement *Value* so that one and the same list can contain an arbitrary number of values with the overall count of the detector, the single count of the counter or the position of a positioner.

- The OI Server is completely decoupled from its range of use. Adding new *Interface Domains* opens up new independent fields of application. Examples are *Interface Domains* for astronomical applications, industry control or *Smart Living* scenarios.

If a new *Interface Domain* is to be published as a replacement for an old domain, it is complex and costly to adjust all modules to the new interface standard. The solution here is to publish an additional OIM that provides mechanisms that act as a bridge between the standards. Both *Interface Domains* will be loaded parallelly in the same assembly while the bridging OIM translates the interfaces, calls and data between the two standards.

4.3.4. The Container Operation Internals

After the previous chapters, which dealt with the development of *OI Assemblies* from the perspective of the user, this chapter gives an insight into the internal processing of an *OI Assembly* by the container. Thanks to the container, a variety of complex tasks, which are otherwise carried out in a manual way, can be automated and hidden from the user. Due to the increased code complexity of these background tasks, this chapter gives only a short overview rather than a detailed description. More detailed information can be found next to the sourcecode under the package *org.openinspire.kernel.container*.

For improved maintainability, all functions required to start and stop *OI Assemblies* have been separated into individual steps, which are processed by a state machine. This eases extending the container by new process steps, analyzing errors and querying the container state. The control and monitoring of the current container state is therefore realized by a service[41] that can be accessed locally or through external network based clients. Figure 4.21 shows the ten steps, which are processed by the container during each start of an *OI Assembly*.

Container Startup

The next sections provide a brief overview and a selection of simplified activity charts for the individual steps, processed by the *OI Container* during the startup of an *OI Assembly*.

[41] More details to the containerservice will be covered in paragraph 4.4.1 on page 141

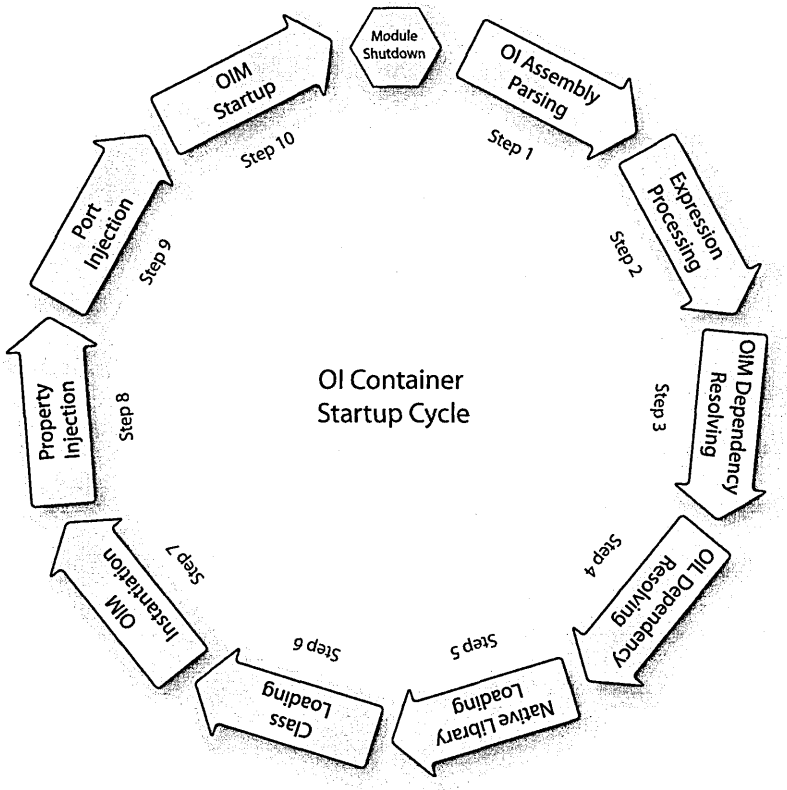


Figure 4.21.: *OI Container Startup Cycle*

Startup Step 1 : OI Assembly Parsing

In the first step, the *OI Assembly* will be opened to be processed by the XML Parser JDOM [jdo]. To detect structural- and syntax errors preliminarily to the logical processing, an XML Schema based validation will be performed[42]. When the validity has been approved, the specification and all OIM tags are read from the assembly and cached as internal object structure for all further processing.

Startup Step 2 : OI Expression Processing

Step 2 is responsible for the processing of Regular Expressions. All ports of the OIM entries, which have been cached during the last step, are progressively reviewed for the presence of wildcards. For each found wildcard, a list of all matching OIM inside the currently cached assembly will be created. As an instantiation is not yet possible, only the module IDs are

[42]If no Schema is given in the assembly, the XML Schema *oi-assembly-1.3.xsd* is used as fallback

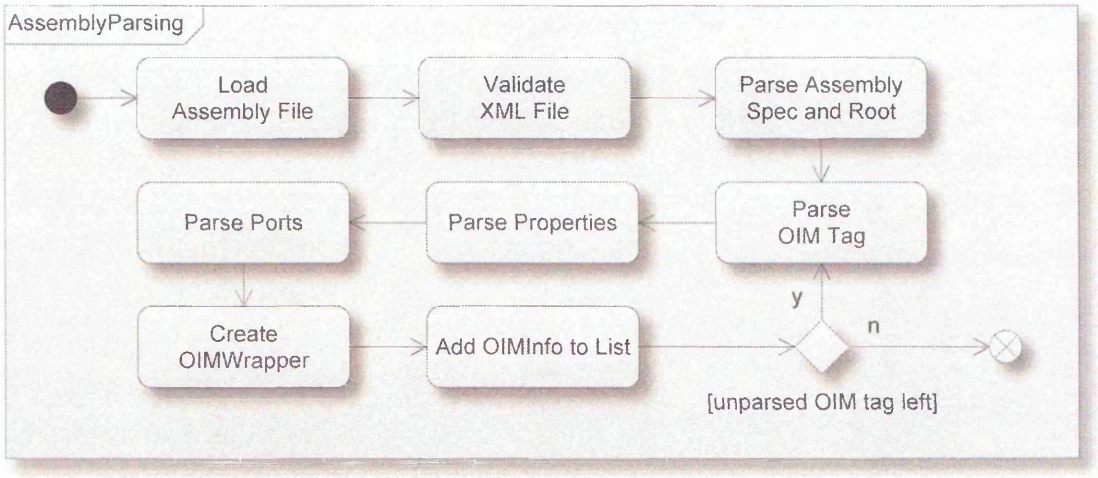


Figure 4.22.: *Startup Step 1 : Assembly Validation and Parsing*

added for further processing in step 9. Since more than one ID can match a wildcard, an automatical distinction between single and multiple references, as covered in paragraph 4.3.2, will be carried out. Duplicates and the module that provides the port are filtered.

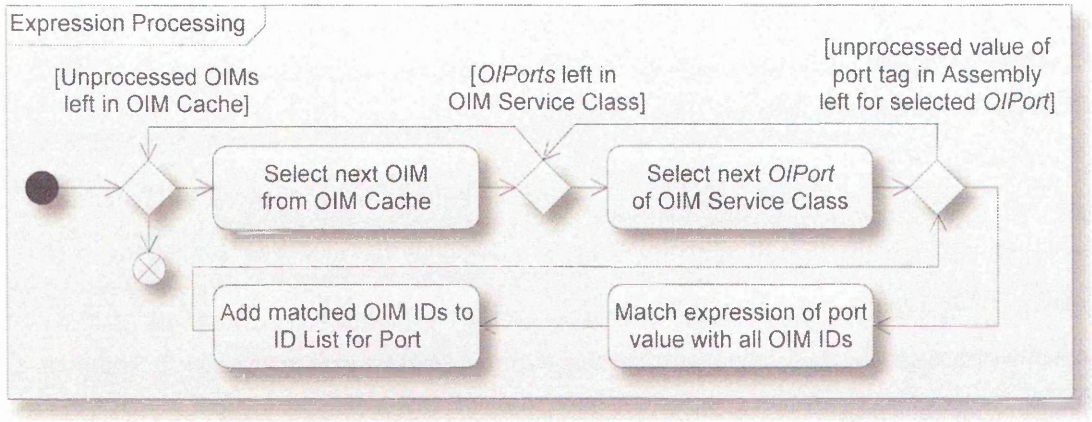


Figure 4.23.: *Startup Step 2 : Regular Expression Processing*

Startup Step 3 : OIM Dependency Resolving

Step 3 checks whether all required OIMs are already installed and rectifies missing dependencies. A list will be created with all installed and another list with all required modules. The *OI Module Installer* subsequently calculates all dependencies of the required modules

that are already installed and downloads the modules from the OI-LAMP that are required but not installed.

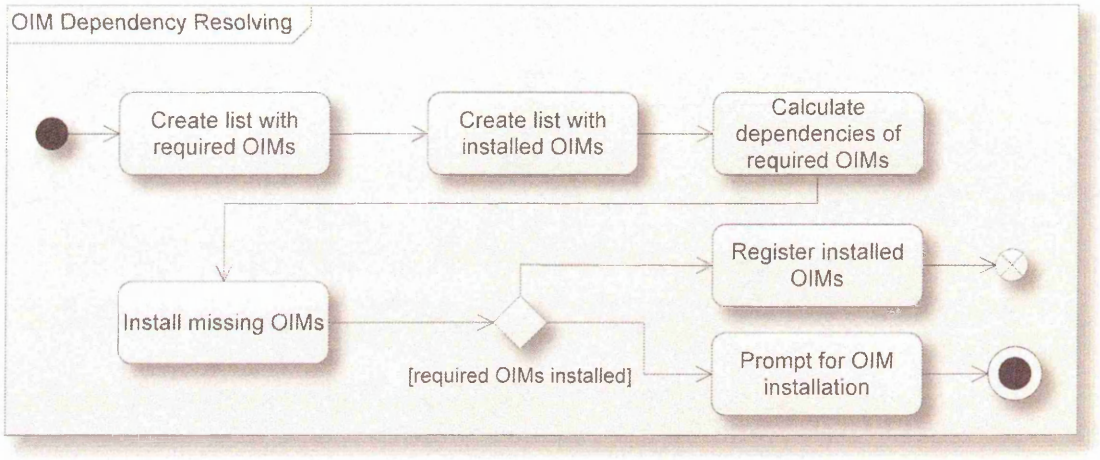


Figure 4.24.: *Startup Step 3 : OIM Dependency Resolving*

Downloaded OIMs can have dependencies again, which will be calculated and downloaded until all dependencies are recursively resolved. Once all OIMs are available, they are registered for all further processing by step 6 and 7. If no network connection is available, the user will be prompted to manually copy all missing modules to the local OIM folder[43].

Startup Step 4 : OIL Dependency Resolving

Step 4 is responsible for the installation of missing library dependencies. At first, a dependency list will be created by aggregating all dependencies that are registered under the *OIM-Lib-Deps* manifest entry of the required OIMs. Since OILs can also have library dependencies, a further scan recursively adds missing dependencies of all required OILs that are installed. If modules are missing, a download of the missing OILs from the OI-LAMP will be initiated. These steps are repeated until all dependencies are resolved. If a module cannot be installed from the OI-LAMP[44] it needs to be installed manually. The container therefore prompts the user to copy the module to the `lib`-folder, detects and uncompresses them for the installation.

[43] These can be modules that have been manually downloaded from the OI-LAMP on a different computer or custom modules that have not been published by the OI-LAMP.
[44] For example modules that have not been published or because of a missing internet connection.

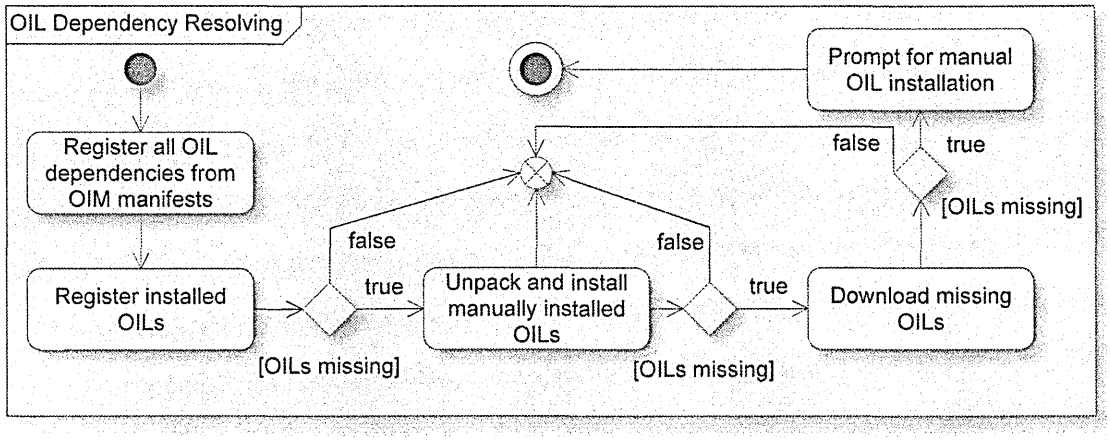


Figure 4.25.: *Startup Step 4 : OIL Dependency Resolving*

Startup Step 5 : Native Library Loading

Chapter 4.2.2 (page 103) introduced the fact that OI allows the use of third-party functionality written in other languages such as *C/C++*. Java provides the Java Native Interface (JNI) for this purpose, which, however, causes various problems and is not directly usable with the strictly modular OI component system. To solve these problems, an extended approach has been developed explicitly for OI, which will be explained in more detail. The main problem is the loss of platform independence, as any native library can only run on a particular architecture and operating system. The second problem is that even if the correct library is available, it needs to be installed in the OS-specific library path before the Java Virtual Machine (JVM) is started. Installing all libraries in advance is however not an option because it is not clear when the OI Server starts, which library dependencies will be required later as requested by components and dynamically loaded by the container. One solution that might be supposed is to prompt the user for a manual installation and restart of the OI Server when missing libraries are requested. In addition to flexibility and usability limitations, this also leads to the problem that it is not possible to verify if the version of the installed library is compatible with the required OIM[45]. At first glance this problem seems to be solvable by adding all required native libraries to the OIL and directly loading them from there. But

[45] And since it is installed globally, only one version can be installed, whose version is probably not compatible with all installed OIMs that make use of the library.

this fails when a library itself has dependencies to other libraries and tries to load them, since Java only makes libraries accessible that were present in the global library path at JVM startup. Dynamic preloading of libraries in an order that accounts for the dependencies does not solve the problem either.

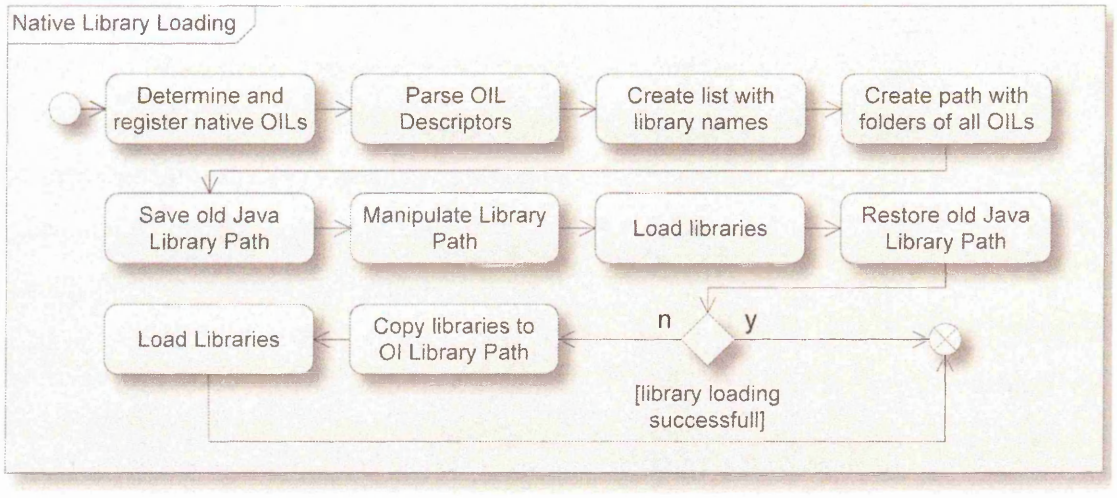


Figure 4.26.: *Startup Step 5 : Native Library Loading*

Nevertheless, in order to remain compliant with the OI design principles, the two-stage mechanism in figure 4.26 has been developed. Defining an OIL as dependency in an OIM manifest causes the container to first search for a platform independent OIL. If no OIL is found locally or in the OI-LAMP a second search looks for a platform dependent version that matches the OS and CPU architecture of the executive system. If a native module is found in the OI-LAMP that matches the naming conventions, introduced in chapter 4.2.2 (page 104), it will be downloaded and installed. Since *Java* does not allow the changing of the preset native library path of the JVM once it is started, a hack is used to overwrite the access rights of restricted JVM code at runtime by means of the *Reflection* API. The container is thereby able to save the original library path, manipulate it to include the loadable native libraries of the OIL, load them in the correct order[46] and finally restore it. A possible reason why *Java* does not allow later changes of the library path are various special cases and OS designs, such as *Windows*, where loading libraries from the manipulated path fails. OI provides a fallback for these situations and copies all native libraries to an OI specific path, which has been

[46]Native libraries are now able to load dependent library since all libraries are temporarily in the library path.

globally added prior to the OI Server start, loads the libraries from there and automatically deletes them when the OI Server or JVM shuts down. The fact that the fallback variant is slower and consumes more memory makes the first variant the preferred method.

Startup Step 6 : Class Loading

To access the resources or classes provided by OIMs and OILs they need to be loadable by the JVM. For a standard Java program the classes are typically added to the application's class-path prior to the start, which makes all resources globally accessible within the program. Analogous to the native library loading in step 5, predicting the classes and resources required by a yet to be started OI Assembly is not possible. Beyond that it is not allowed to use multiple versions of the same class in one application due to the lack of class versioning mechanisms.

OI solves these problems by the three new class loaders *OIMClassLoader*, *OILClassLoader* and *OISystemClassLoader*. Each type of OIM and OIL that is used directly or as dependency in the current OI Assembly gets assigned a dedicated class loader that gives access to the resources and classes provided by the specific module. If an OIM tries to load a class or resource, it first considers its own *OIMClassLoader*. If the file cannot be found, the loader delegates to the OIM and OIL ClassLoaders of the modules, which are set as dependency. Each module can in turn have its own dependencies so that a chain of class-loaders is evaluated until the resource is found. The complexity here is that OI needs to take care that each module gets access to the correct version of dependent libraries. If an OIM A owns a dependency to an OIM B and an OIL C, and OIM B itself has a dependency to OIL C with a different version, it needs to be ensured that all classes in OIM A use version 1 of OIL C and the classes of OIM B use version 2 of OIL C.

SECURITY: The introduced mechanism fulfills all security requirements [☒ 3.2.5][☒ 3.2.3] since each module is executed within a shielded environment and can exclusively access resources in its own module boundaries or that are set as dependency. Accessing or compromising code of the server or modules that are not set as dependency is prohibited. Only a secure

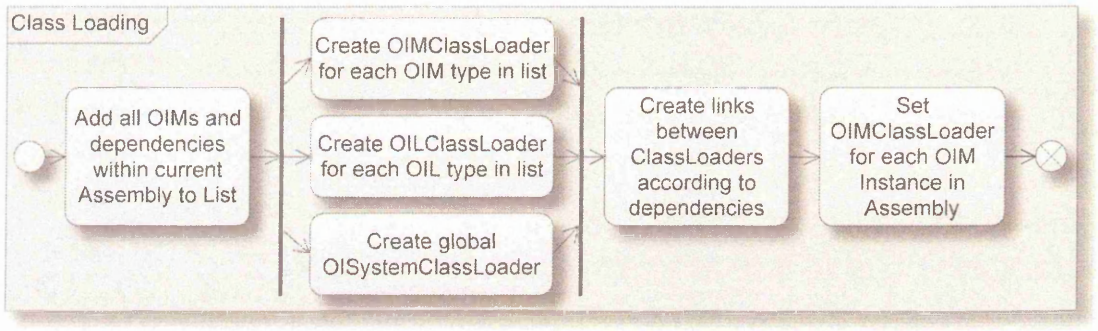


Figure 4.27.: *Startup Step 6 : Class Loading*

selection of global container services[47] are made available by the *OISystemClassLoader* that also controls the module access to the Java API.

PERFORMANCE: Analyzing the classes and resources for each module and its dependencies is slow, when it is performed every time a resource or class is loaded. To speed up this process, the *OIClassLoaderCache* caches all data at the first time and uses efficient hash algorithms for subsequent access to the data pool.

Startup Step 7 : OIM Instantiation

Since each OIM is, in the meantime, able to load resources and classes with its assigned class loader chain, the startup continues with the instantiation of the OIM Service Classes. The container therefore opens the manifest of each OIMs in the Assembly, reads the name of the *OIServiceClass*, loads the class with the assigned *OIMClassLoader* and creates an instance.

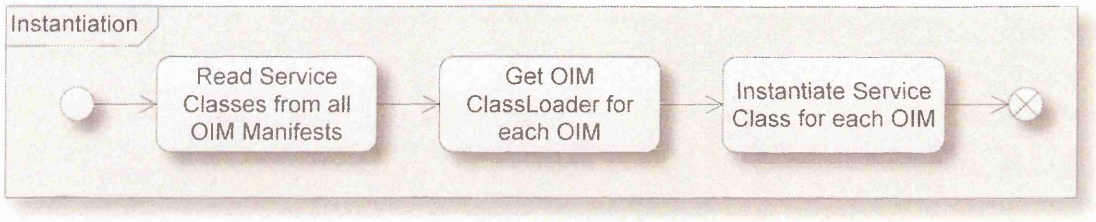


Figure 4.28.: *Startup Step 7 : OIM Instantiation*

[47] An introduction to the Global Services follows in section 4.4

Startup Step 8 : Property Injection

After all OIM instances are created, the container continues with the injection of the OIM specific properties specified in the OI Assembly.

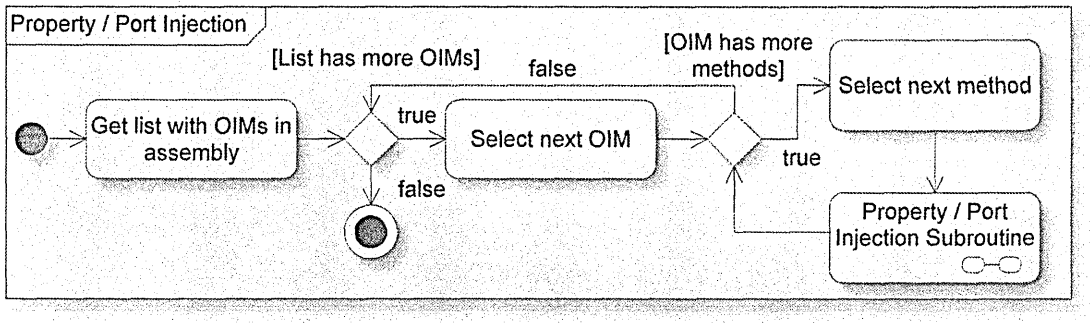


Figure 4.29.: *Startup Step 8 and 9 : Property / Port Injection*

The Reflection mechanism is used to scan each OIM Service Class in the assembly for methods that begin with *set*, have a single parameter and are annotated with *OIProperty*. Matching methods are then called with the appropriate *property*-value for the specific OIM in the Assembly. If a matching *property*-entry is missing no injection will take place. If the *mandatory*-attribute is set for the *OIProperty* in the source code, the startup will be canceled after printing an error message, otherwise it will be ignored and no injection takes place for the *property*. OI is able to interpret the text entry of the *property* and casts the specific value to the String, floating point or integer value, required by the method. Figure 4.29 and 4.30 provide an overview for the *Property Injection* process.

Startup Step 9 : Port Injection

The *Port Injection* step is responsible for the interlinkage of modules. Similar to the *Property Injection*, all methods of the Service Classes are searched for *Setter*-methods with a single parameter but this time with an *OIPort* annotation. Since linking more than one OIM instance to a port[48] is allowed, a first check determines if the method's parameter type derives from *Collection* and supports multiple references. If this is true, it is possible to compose the appropriate *Collections* with help of the values that have been aggregated during the

[48]Multiple references are based on Java Collections and covered in section 4.3.2 on page 117

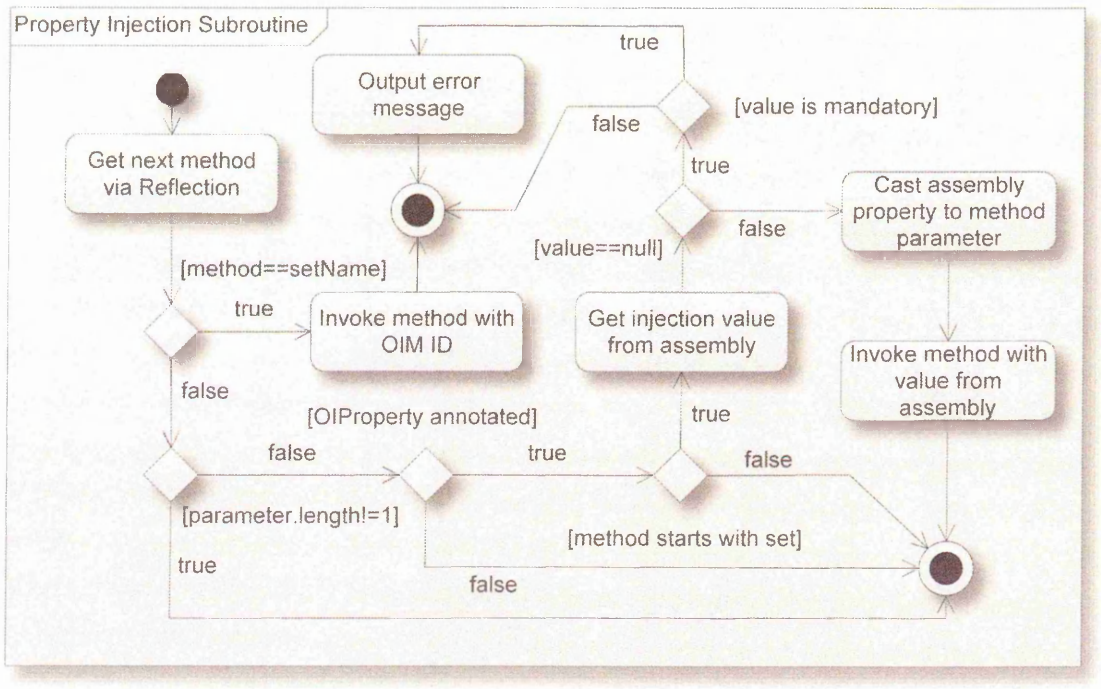


Figure 4.30.: *Startup Step 8 : Property Injection Subroutine*

evaluation of wildcards and multiple port entries in step 2. However, in contrast to the *Property Injection* not the values themselves but the OIM instances are injected. These can be obtained from an "OIM ID ↦ OIM Instance" mapping, created in step 7. OI then calls all *OIPort* annotated OIM methods with the newly created *Collections* of instances. Using *Collections* is admittedly a special case and only used for multiple references. For single references a simple injection takes place where the port value is read from the *Assembly*, replaced by the OIM instance and passed to the *OIPort* method. OI automatically casts the OIM instances and creates *Collections* of the correct type to match the input parameter of the *OIPort*. *Mandatory*-tags are supported and handled in the same way as in step 8.

Startup Step 10 : OIM Startup

Step 10 is finally responsible for the initialization and startup of OIMs. Normally all initialization code for *Java Classes* is placed in the *constructor* of a class and called as part of the instantiation. This also works for OIMs that have no *OIProperties* and *OIPorts* but fails when OIMs need to be either externally configured or require references to other OIMs.

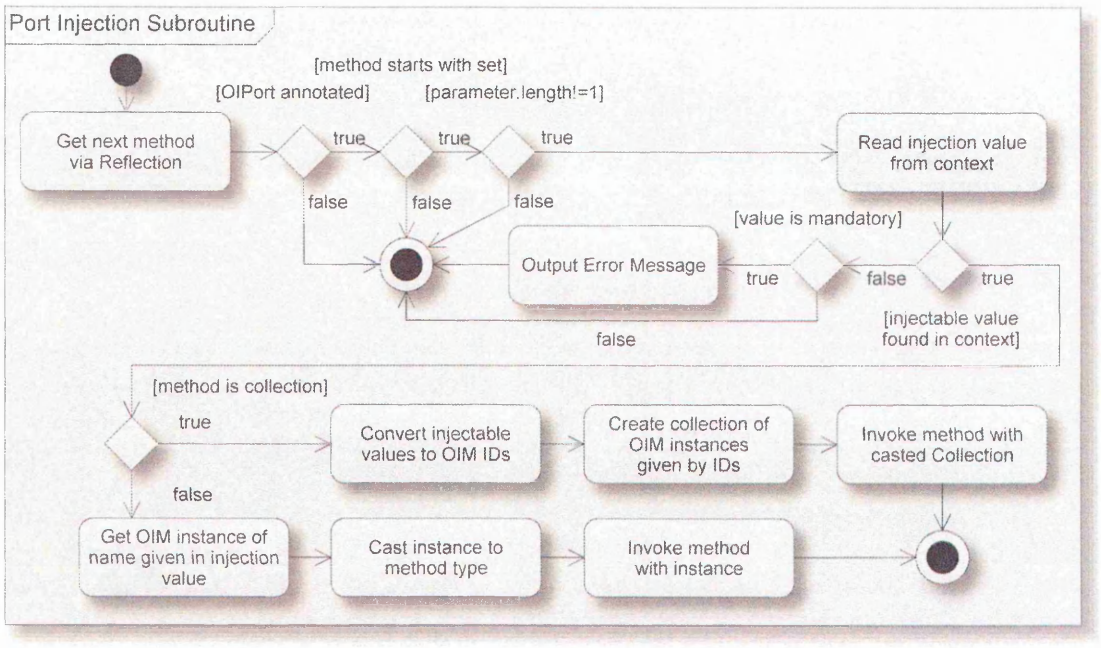


Figure 4.31.: *Startup Step 9 : Port Injection Subroutine*

The reason is that these values are first injected in step 8 and 9 after the constructor has already been executed[49]. The solution is the method *initComponent()*, which can be optionally added to OIM *Service Classes* and is triggered after the injection process, when all *properties* and *ports* are accessible. The challenge here is to call the init methods in the correct order. Starting an OIM before all modules are initialized, which are connected to the OIM's port, can provoke crashes caused by uninitialized objects. OI solves the problem using topological sorting algorithms that create trees of all active OIMs and dependencies, resolve cyclic dependencies and determine the correct OIM startup order. Details to the algorithms can be found in the *oi-system* library and are due to the complexity not part of this overview.

Container Shutdown

Shutdown Step 1 : OIM Shutdown

The step *OIM Shutdown* is used to cleanly shutdown an active *OI Assembly.* Each OIM *Service Class* can, as counterpart to the *initComponent()* method, come with an optional

[49]Injections are only feasible on objects so that it needs to be performed after the instantiation in step 7

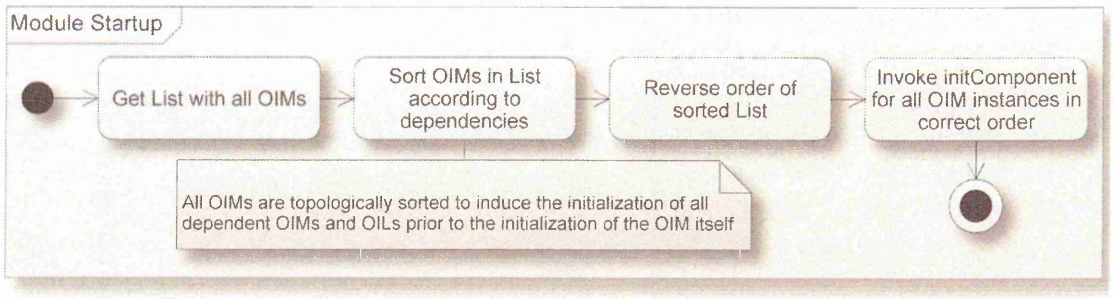


Figure 4.32.: *Startup Step 10 : Module Startup*

method *shutdownComponent()*. This can be utilized by OIM developers for cleanup tasks such as closing files and network connections, stopping threads or freeing memory. If the container finds such shutdown methods, it respects possible runtime dependencies and calls them in reverse order than in step 10. After completing this voluntary OIM cleanup process, the container memory is freed and the server will be prepared for the start of new *Assemblies*.

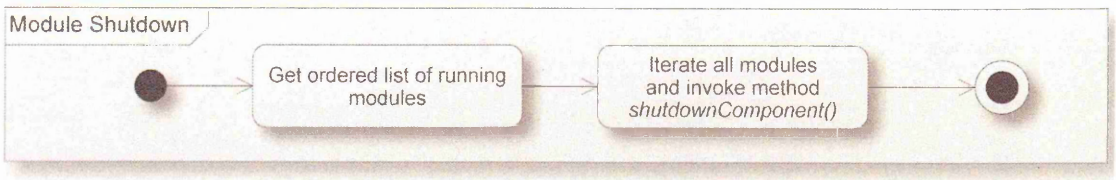


Figure 4.33.: *Shutdown Step : Module Shutdown*

4.4. Services and Configuration

The preferred way to provide a service by OI is to publish and operate it as OIM within the OI Container. This is certainly only possible for services that are not used for the server management itself and hence need to be independent from the container.

In order to provide services that are not bound to the container state, OI comes with a *Service Registry* that allows registering global services. These standard services dock against fixed and predetermined interfaces and are hence easily replaceable by other implementations.

4.4.1. Shared System Services

The following section provides a brief insight into the key OI Services: *Authentication Service*, *Filesystem Service*, *Container Service*, *Network Service* and *Logger Service*. Figure 4.2 on page 79 already showed the placement of these services within the OI Server.

Authentication Service

The *Authentication Service* is used for the authentication of users by username and password. A successful authorization allocates the user to a role and unlocks the access to specific resources and extended functionality [☒ 3.2.8]. The service is primarily used to restrict the file access to the OI Filesystem *(FileSystem Service)*, to prevent unauthorized changes to the container state *(Container Service)* and for the login to the OI *Shell*[50]. Another object is the handling of role-based security mechanisms within the *OI Assembly*, introduced on page 120.

The configuration of the privileges is managed by the XML file *privileges.xml*[51], which contains the two main sections *roles* and *users*. *Roles* is used to configure an arbitrary number of roles, which have an *id*, a *description* and can inherit the privileges of other roles. By default, the roles *admin*, *user*, *observer* and *shell* are predefined. The role *admin* thereby inherits all rights from *user*, which in turn inherits the rights from *observer* and *shell*. Adding new roles enables the easy and flexible creation of complex privileges networks. The other section *users* finally provides a list of users with a unique name, an encrypted password, a description and an assigned role. Here the users *admin*, *user* and *observer* are predefined and assigned to the correspondent role [☒ 2.2.2][☒ 2.2.3].

Filesystem Service

The *FileSystem Service* (FS Service) provides access to *Assembly* files and stored OIM data such as measurement results. This makes external applications and modules not only able to read and write server-stored data but also allows the copying, moving, deletion and renaming of files within the directory structure, or the setting of extended attributes such as file locks. The service acts as an apparently transparent barrier between filesystem and service

[50]The Open Inspire Shell is a Command Line Interface (CLI) covered on page 143.
[51]See graphics 4.4 on page 84 for the default location of the *privileges.xml* in the filesystem.

consumers. It is therefore the point where transparent filtering mechanisms such as the role-based access control or automatic file history and version management are implemented. For the access control it works closely together with the *Authentication Service* and for external clients it is made accessible by the *Network Service*. Network clients primary use the *FileSystem Service* to create and edit *Assemblies* and to observe or download measurement data[52]. An optimal throughput and bandwidth consumption of transfers is realized by streams while asynchronous notification mechanisms keep simultaneously connected clients synchronous.

Container Service

The *Container Service* is used to control and query the status of the OI Container. This primarily includes starting and stopping OI *Assemblies* as well as setting the current *Assembly* using a path and filename that is managed by the *FileSystem Service*. Beyond that, it is possible to read the container state, progress of the currently processed startup- or shutdown-step as well as messages and error states. Beside the active reading of container states, clients can furthermore register themselves as observer and receive notifications about state changes.

Network Service

OI comes with a *Network Service* that allows publishing services transparently via network. Providing the network support by means of an exchangeable implementation enables the changing or adding of network protocols without touching other OI Services. Moreover it is possible to operate different network implementations at the same time. This allows access to the same data model while communicating with multiple clients using different protocols, which in turn makes OI flexible when new standards arise or the provided implementation turns out to be incompatible with specific architectures[53].

To meet all requirements for a firewall-friendly architecture OI comes with a *Cajo* [caj11] based default network implementation. The solution completely abstains from dynamical ports and requires no open client-side ports for callbacks, which makes it easily and securely tunnelable through existing firewalls [☒ 6.5.2]. All relevant network functionality including

[52] Each OIM has a dedicated storage for data such as measurement results that is accessible by the FS *Service*

[53] An example is the smartphone operating system *Android*, which comes with a reduced JVM that lacks RMI support and is hence incompatible with the OI default network service.

secure tunneling through encrypted *SSH* connections is provided by the *oi-system* library. Establishing a remote connection is done with one command that only requires the hostname and port of the network service and returns a *Lookup* with all free services provided by the *OI* server. An additional authentication with username and passwords expands this Lookup by restricted services that are accessible by the user's role. Since services can be dynamically added and removed, all clients are notified in case of *Lookup* changes.

Registering network services is not limited to global services but is also made possible for OIMs inside the container by adding the OIM *ServiceRegistry* to an OI *Assembly.* All modules that are connected to this registry module automatically publish their service interface using the *Network Service* in due consideration of the access restriction given by the *role*-tag. Clients can access the published interfaces in the same way as local services and thus use it for example as data provider for widgets. GUIs are thus able to dynamically activate or deactivate controls, menu items or other UI-elements when services are added or removed from the *Lookup*[54].

Logging Service

With the *Logging Service* another important service is available, which is primarily used by the OI Server and OIMs inside the OI Container. It is a convenient interface to the messages generated during the runtime of Open Inspire and thus an important instrument in meeting the stipulated error analysis requirements [☒ 6.4.2]. The service collects all log messages that are generated for debugging or informational purposes using the standard *java.util.logging.Logger* command in the OI Server, OIMs, JVM or external libraries and saves them to the file *inspire.log.* For a reduced file size and improved handling, it is a rotating log file, which means that a new empty file is generated every day and the old file is saved with appended date. Each log entry contains the date and time of the message, the class and method in which the message has been logged, the log level[55] and the message itself [☒ 6.4.3]. Error analysis-, profiling- and visualization-tools can either access the local log file or read it as stream from the service.

[54] More details about the dynamic activation of UI elements follow in section 5.1.1 on page 176

[55] Valid log levels are *SEVERE, WARNING, INFO, CONFIG, FINE, FINER* and *FINEST.* More details can be found directly in the configuration file, which is covered in 4.4.3 on page 145.

4.4.2. User Interaction Services

Since a large number of users should be addressed, their preferences in relation to the UI need to be taken into account. OI provides a set of services that can be used as backend for various forms of User Interfaces. Three UI realizations are already implemented and discussed below.

Network based User Interface Clients

The recommended way to operate the *OI* server is in headless mode as background service because this guarantees an optimal decoupling between data model and view. As long as a GUI client has access to the OI *Network Service*, there are no limitations on the shape or implementation of a client. Clients in *Java*, *Groovy*, *Scala*, *Jython* or other languages based on the JVM can make use of the *oi-client*-library, which already provides all functionality for the authentication, communication, remote access, encryption and tunneling. Clients in other languages can either directly access the default protocol or implement a new *Network Service*. More detailed application examples follow in chapter 5.1 on page 175, which introduces the *RemoteLookup* of the *Network Service* and several UI implementations.

The Open Inspire Shell

Beside the headless or daemon mode, it is possible to start OI in the *Shell* mode, which allows the operation of Open Inspire using a Command Line Interface. The OI *Shell* is platform independent and thus works on *Mac OSX*, *Windows*, *Linux* and other *UNIX* derivates in the same way. Compared to the network based coupling of clients, the *Shell* liaises more closely with the system core and can hence provide some extra features like enhanced debugging mechanisms.

If OI is configured to start in *Shell* mode, it shows the start screen pictured in Figure 4.34 and prompts for a username and password. If the user can successfully authorize himself as a member of the role *shell*, he obtains access to a Command Line Interface. In the special case that the authentication is disabled in the OI configuration, the *Shell* reclaims prompting for a login.

```
auriga:release stefan$ ./start.sh
  ____                   ____                  _
 / __ \___  ___ ___     / _/__  ___ ___  (_)_____
/ /_/ / _ \/ -_) _ \   _/ // _ \(_-</ _ \/ / __/ -_)
\____/ .__/\__/_//_/  /___/_//_/___/ .__/_/_/  \__/
    /_/                           /_/
          Server Shell 1.3.0_alpha7     (Type help)
.................................................
 (c) Stefan Alexander Flemming www.openinspire.org
.................................................

Login: admin
Password:

OI-Shell => []
```

Figure 4.34.: *OI Shell Login*

The use of the *Shell* is almost self explanatory. A good starting point is the command *help*, which outputs a list of all available commands (figure 4.35). For further information, *help* in combination with a command name gives a detailed description of a specific command.

```
OI-Shell => help

Open Inspire Shell Commands:
-----------------------------------------------------
asm         Lists, selects and starts assemblies
exit        Shuts down the Open Inspire Server
help        Prints help or command description
log         Prints the content of the logfile
oil         Shows information about OI Libraries
oim         Shows information about OI Modules
quit        Shuts down the Open Inspire Server
services    Lists all active services
sysinfo     Prints system environment variables
version     Prints OI version and build information

OI-Shell => []
```

Figure 4.35.: *OI Shell Help*

Adding new OI *Shell* commands is straightforward by internally registering a class that implements the *Command*[56] interface. This interface comes with the methods `execute(String[] parameters)`, `getShortDescription()` and `getLongDescription()` that need to be overwritten with methods that provide the command logic and descriptions for the *help* command.

[56]The interface can be found in the OI Kernel sources under *org.openinspire.kernel.ui.shell.cmd.Command*

The Open Inspire System Tray

The third default User Interface provided by OI is a graphical UI that makes use of the *SysTray*. If it is enabled in the OI configuration, a new icon appears in the *System Tray* of the Operating System and provides access to a configuration and control menu. Opening the menu shows items that allow the server to be halted, the starting and stopping of the *Assembly* and selection of available *Assemblies* from a nested tree as shown for *Mac OSX* in Figure 4.36.

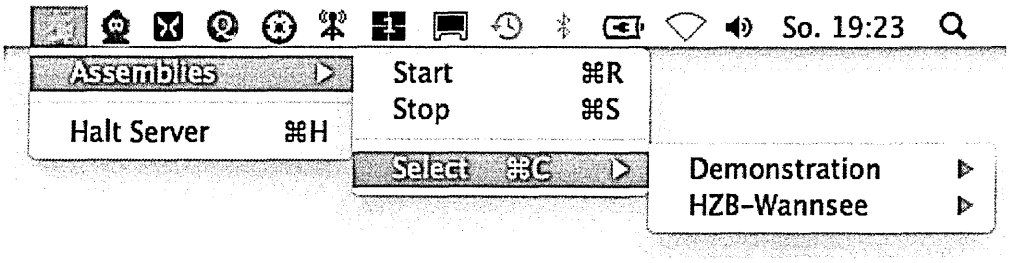


Figure 4.36.: *Mac OSX System Tray*

The *SysTray* menu thus meets the requirements for providing an easy means to operate the OI server for systems such as *Windows*, where using a CLI is unusual. Open Inspire is hence startable as a background service and though remains operable without the need for external clients.

4.4.3. The Service Configuration

OI comes with a configuration file that makes the main behavior of the OI Server configurable and allows setting up OI according to the good practices of Operating Systems or specific user demands [☒ 2.1.5][☒ 6.1.3]. It is named *inspire.properties* and per default located in the *conf*-folder[57], beside the already known *privileges.xml* for the configuration of user rights and roles. The file is divided into the five segments *Network*, *User Interface*, *Logging*, *Storage & Persistence* and *Update & Repository*, adequately commented and self-explaining so that no detailed description but only a brief overview of the main configuration sections is given in table 4.3.

[57] See graphics 4.4 on page 84 for the location in the filesystem

Section	Description
Network	Allows setting the IPs *Address* and *Ports* of the *Network* and *Registry Service.*
User Interface	Allows to set if OI should whether start as a background service or interactive shell, if the *SysTray* should be enabled and which Look and Feel (LnF) to use. Currently the LnFs *System LnF*, *Metal* and the modern LnF *Nimbus* can be set for the decoration of server side dialogs.
Logger	Allows setting the path and filename of log file as well as the *LogLevel.*
Storage and Persistance	Allows to configure the storage root for OIM data and the *cache*-directory. The *cache*-folder is used to store temporary data generated by the module installer, as cache for native libraries and as a buffer for OIM data. The file *cache.properties* is automatically created within the *cache*-folder and used for data that should persist system restarts. The OI server can hence remember the lastly started *Assembly* or the last startup date.
Update Repository	Allows to replace the default URL of the OI-LAMP *Update Center* by a local mirror or overlay.

Table 4.3.: *The OI Configuration File*

To comply with the standard paths of different Operating Systems, it is possible to start the OI server with the parameter *config*, that expects the path to the *inspire.properties.* Respecting the *Linux* Filesystem Hierarchy Standard (FHS) [Qui94], for example, is thus possible by adjusting the paths within the *inspire.properties* and relocate the configuration files to */etc*, write the logger's output to */var/log* and start OI using a startup script.

4.5. OI Library and Media Platform

An important attribute of Open Inspire is that it is not a local and foreclosed system but comes with multiple collaboration features that ease the global exchange of information. The center of the system forms the publicly accessible component repository at `http://store.openinspire.org`, which has been scratched in Section 4.1.1 (page 76).

While this repository is used by several automated OI services, which access the globally published OIMs, OILs and OI *Assemblies*, it is not suitable to be used as human readable

information source. To meet all collaboration and learning requirements it is therefore necessary to provide additional services to ease the exchange of information between users and allow quick access to the required documentation and resources [☒ 5.2.2][☒ 5.2.3][☒ 5.2.4][☒ 2.3.9].

4.5.1. The Project Webpage

The website `http://www.openinspire.org` is the entry point for everyone who is interested in or working with Open Inspire. It is designed in a uniform style and optimized to be viewed on high resolution screens as well as the small displays of tablets or smartphones[58][☒ 1.1.2].

Figure 4.37 shows a snapshot of the site. The head is a dynamically adjustable menu with links to the sections *Home*, *Documents*, *Collaboration*, *Assemblies*, *Modules* and *Downloads*. The left side contains a submenu that shows the entries of the currently selected main menu section and the third menu under the main menu is a *breadcrumb* that indicates the path to the current webpage. A password-protected backend allows editing all static content, the integration of dynamical services and setting up user access privileges.

Figure 4.37.: *Snapshot from the OI Website*

[58]Modern smartphones provide in landscape mode enough space to display the complete width of the page.

To reduce the administration effort and ease the publishing of content, a Content Management System (CMS) is used. From the shortlist of the CMSs *Typo3*, *Joomla*, *Drupal*, *Wordpress* and *Magnolia*, *Typo3* [typb] has been chosen. The *Open Source* system *Typo3* comes with a configurable backend that i) allows the creation of pages using a rich text editor, ii) takes over tasks such as the user and access management and iii) can be extended using plugins (important to allow the integration of OI specific dynamic content). The drawback of this system is that it is written in *PHP* and thus not as easy integrated in the OI environment as a *Java*-based *CMS*. But this is compensated by *Typo3*'s popularity. Thousands of pages of organizations such as *Airbus*, *DHL*, *EADS*, *UNESCO*, the HZB, BHT, *Greenpeace*, *Lindt* or *T-Online* all commit on *Typo3* [typa]. This makes it very likely that the development and support will be continued.

4.5.2. Content and Backend Services

For a rapidly changing and developing project, such as Open Inspire, it is difficult and time-consuming to keep all documentation, links and services up-to date. As a result, the time effort rapidly shifts from the development of the web infrastructure to permanent maintenance. To meet the requirement that all published content is up-to date [☒ 3.8.5] and that the development is not hindered by maintenance tasks, a dynamic service infrastructure is used that automatically updates the web pages when sources, components, assemblies or binary *OI* packages change.

Figure 4.38 illustrates the *OI Service Infrastructure.* To meet the requirements regarding the reuse of existing systems [☒ 5.3.6] it incorporates several third party products. These have been made to communicate with each other and integrate seamlessly into the OI environment. The following paragraphs give a survey to the website's main sections and the integrated services.

Documentation

The section *Documentation* contains primarily static content and is divided into the four subsections *Manuals*, *Tutorials*, *Screencasts* and *API Documentation.* These subsections provide information that is addressed to beginners as well as advanced users and programmers.

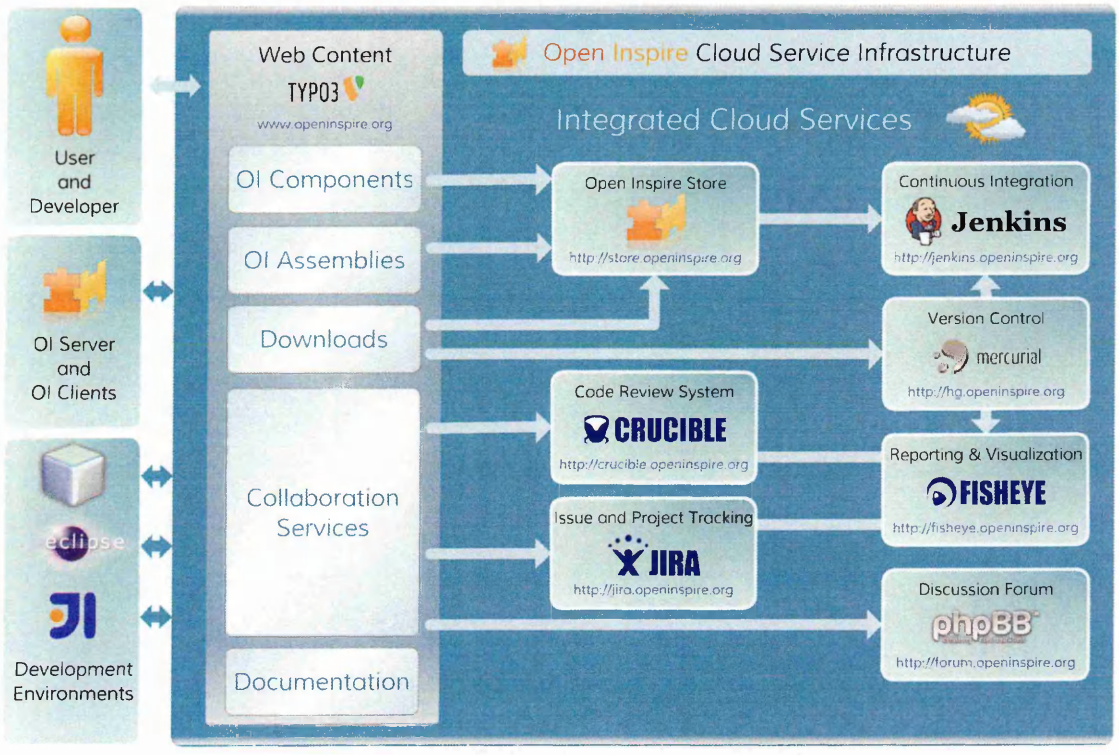


Figure 4.38.: *The OI Web and Cloud Infrastructure*

Screencasts are the preferred medium. The section contains short videos, used to introduce a user into a specific topic. The advantage of a screencast is that a user can exactly reproduce each step due to the fact that each interaction with the computer such as mouse movements, entering text or opening a menu are directly obvious. For the creation of screencasts the software *Camtasia* [com] has been selected since it is available for *Mac OS* as well as *Windows* and comes with utilities such as the highlighting of screen cutouts, zooming and the embedding of videos and speech.

An addition to the *Screencasts* are *Tutorials* that use short texts and illustrations to introduce functionality step by step. *Tutorials* can be printed and consume less hardware but are not as exact in their description and clearly reproducible as *Screencast*.

The third document type are *Manuals*. These are more comprehensive and not aimed to users that want to learn functionality but for everyone who search more detailed information.

The last category is aimed to developers and contains the OI API and source code documentation. The OI source code is commented and contains descriptions, which makes it possible to automatically create a human readable code documentation. The documents contain descriptions of all packages, classes, methods and attributes as well as their dependencies among each other. The generation of the documentation occurs automatically when sources inside the OI Version Control System (VCS) change. As generation tool *Doxygen* has been selected, which comes with several advantages over *Java's* standard documentation tool *javadoc* such as the creation of PDF documents and more sophisticated configuration switches. *Doxygen* is embedded into the *OI Build System* and automatically triggered by the Continuous Integration System (CIS), which will be covered in connection with the *Download* Section.

Components

The Section *Components* provides user-friendly access to the OIMs and OILs that are published using the OI Store. While the store is primarily used autonomously from the OI *Server* and *Clients* for the downloading of components or querying of meta data, the *Components* Section makes these and some additional features accessible through human readable webpages. Functions include the ability to browse the catalog of published components, query the component descriptions, submitting new components, downloading components or adding comments and votes[59]. Beyond that, it is possible to generate OI *Components* by means of a step-by-step wizard. This wizard asks for the features that a module should have and creates a package with all necessary meta files and preconfigured *Java* classes including the annotated method bodies for the OI *Ports* and OI *Properties.*

OI Components are written in *Java*, so that the *OI Store* also uses *Java* to process the modules' meta information such as the values from the OIM *Manifest* or the OI *Port* and OI *Property* annotations. Embedding the *Java*-based store into the *PHP*-based *Typo3* website is difficult so that the store provides an independent *web service* that is used by the OI *Server*, OI *Clients* and the website to query all relevant information remotely.

[59]Commenting and voting for components allows users to give a quality and popularity feedback

Assemblies

The Section *Assemblies* is similar to the *Components* Section but provides access to the catalog of globally published OI *Assemblies*. Analogously to the components catalog the *Assemblies* section is split into categories and comes with a simple search engine that helps finding a desired *Assembly*. The catalog entries show the name, description, all meta-information, the list of dependent components and a unique *ID*. This *ID* is used for a feature called *Rapid Assembly*. Passing the *ID* to the server[60] triggers the download and installation of the corresponding *Assembly* and all dependent modules and libraries. One short *ID* is thus all that is required to assemble the complete setup of an experiment. Such an *ID* can subsequently be published with papers or shared between experimenters.

Downloads

The fourth of the seven website's main sections is the *Download* area. From this section it is possible to download the OI sources, binaries, related applications and resources [☒ 4.4.3][☒ 4.4.5] . An important requirement here was to provide the most recent downloads without distracting the development process due to publishing-tasks. This is done by several background services provided by the official *Mercurial* front-end and the Continuous Integration System (CIS) *Jenkins*.

The dynamic access to the sources is realized by the Version Control System (VCS) at `hg.openinspire.org`, which allows browsing all versions of all OI sources that have ever been committed. Committing an update to the sources automatically triggers the creation of a *bz2*, *zip* and *gz* archive and makes the sources available through the download section.

A difficult and more time-consuming task is the provisioning of the installable OI *Binaries*. Whenever sources change, the binaries need to be rebuilt, using the *Build System* covered in Section 4.6, and repackaged for all supported Operating Systems and platforms [☒ 4.4.2]. Performing these steps every time manually, even for minimal source code changes, hinders the development and has thus been automated using a Continuous Integration System (CIS)[61].

[60] Entering `ra 12.3` inside the OI *Shell* for example installs version 3 of the public *Assembly* number 12

[61] A CIS is a tool that automatically triggers the build process either when sources change or on a regular time base. More datails follow in the next Section. The OI CIS is hosted at `ci.openinspire.org`

Whenever code is submitted, a VCS hook triggers the CIS *Jenkins*, which subsequently gets all sources from the VCS, configures the OI *Build System* and builds a platform independent OI release [☒ 4.3.3]. This release can be used to run OI without installation on OSs where no explicit OI *Installer* is available or for the packaging of OI for OSs that come with a custom installation system and download the OI binaries during the installation[62]. To reduce bandwidth and to provide a unique download link, all releases are packed as *zip*-, *tgz*- and *bz2*-archives, tagged with a version, uploaded to the OI store and made available through the *Download*-Section [☒ 4.3.3].

The platform independent packages are sufficient to run OI on all OSs but they do not observe the OS specific characteristics such as system specific paths or the lifecycle-management of services. This is why OI also comes with optimized installers for the most widely used OSs, i.e. *Windows*, *Mac OSX* and *Linux* [☒ 4.3.2]. The main difficulties here are that these operating systems are very different in their usual installation mechanisms and that the installer binaries can only be built on the related platforms. To prevent all OSs needing to be installed on the continuous build system (to allow building the binaries for all platforms), a cross-compilation environment has been setup that is able to build binaries for *Linux*, *Windows* and create *OSX PackageMaker* setups.

For *Windows*, the Nullsoft Scriptable Install System (NSIS) [nsi11] is used, since it is *Open Source* and flexibly configurable via configuration files. Such a configuration, which reflects the current version, paths and specific system characterics[63], is generated for each release by the OI *Build System*. *Jenkins* subsequently calls the *NSIS* installer generator to cross-compile[64] a single *Windows* exe using the configuration file, platform independent OI package and additional resources such as icons and images. Starting the generated *Installer* checks if a current *Java* installation is present, downloads and installs *Java* when it is missing, lets the user choose the OI install location, select optional assemblies, modules and documentation and then installs and configures OI for Windows, creating all startup scripts and an optional start-menu and desktop shortcut.

[62] Many *Linux Installation Systems* make use of such an automated download.
[63] For *Windows*, OI is for example configured with an activated OI *SystemTray* (Paragraph 4.4.2, p 145)
[64] The installer executable is created using the Minimalist GNU for Windows (MinGW) [min11] cross-compilation environment under *Gentoo Linux*.

For *Linux* it is unusual to have a binary installer. Here the OI *Build System* creates build configurations or packages for the installation system of the specific *Linux* distributions[65]. OI installation packages that install and build OI from the sources are provided for *Gentoo* and *Sabayon Linux*. The installation downloads the sources, builds OI, copies all files to the OS specific locations and adjusts the OI configuration files to match all file locations and platform characteristics[66].

Mac OSX comes with a tool called *Package Maker*, which creates application packages that behave like an individual file, which can be installed by copying it to the *OSX Application Folder*. The fact that *Java* is pre-installed on *OSX* and that the application package can be configured and equipped with everything required to run OI makes this installation very easy. Building the *Application Package* requires no cross-compilation and it can be directly assembled with *Jenkins* under *Linux*.

To meet all data integrity requirements all files hosted at the global OI Server are backed-up daily and stored on individual servers in independent networks. For improved scalability and fail-safety, the URL `store.openinspire.org` automatically balances the load to the independent mirrors `store1.openinspire.org` to `store9.openinspire.org`.

Collaboration

The last Web Section provides services that help improving the collaboration between users and developers. The main communication is handled by a forum at `forum.openinspire.org`. It is divided into public sections for everyone who is interested in OI and registered users and developers. A forum or a mailing list is the core of a community project, since it allows the collaborative answering of questions. The advantage of a forum over a mailing list is, that all contributions are centrally browseable, searchable and readable and that special formattings[67] and the inclusion of graphics is possible and consistent for all viewers.

Beside the forum, several services are made available that mainly serve to improve the development. The first service is the issue tracking, reporting and project management tool *JIRA*

[65] Common installer systems are for instance the RPM Package Managers (RPMs) for *RedHat*, *CentOS* and *SuSE*, Advanced Packaging Tool (APT) for *Debian* and *Ubuntu* or *Portage* for *Gentoo* and *Sabayon* Linux

[66] On *Linux*, OI is started per default as background daemon with disabled OI *SysTray*

[67] Source code can for example printed in a monospace font and equipped with line numbers.

[jir]. It is accessible at `jira.openinspire.org` and allows the reporting of bugs, feature requests and issues, and the acquisition of the current project state and statistics. Statistics can relate to open issues, the overall stability or timeliness of project development. Issues can be reported, assigned to developers and tracked, so that everyone that subscribes to a bug gets emails with status updates. Furthermore it is possible to connect a bug or issue to code. This is done with help of the the tool *Fisheye* [fis], which acts as a bridge between the OI *Mercurial* VCS with *JIRA*. *Fisheye* is hosted at `fisheye.openinspire.org` and provides sophisticated services such as real-time notifications of code changes, web-based reporting, visualization, search and code sharing. Figure 4.38 shows that not only *JIRA* is using *Fisheye* but also *Crucible*. *Crucible* [cru] is a code review tool that allows the addition of inline comments to source code. A discussion related to issues or code changes can thus be added as layer directly over the code without altering the code inside the VCS itself. This simplification of the development workflow and introduction of asynchronous reviews is also supplemented by different types of change notifications.

The tools *Crucible*, *Fisheye* and *JIRA* have been selected because they interact to perfection and are very popular with more than 26,000 customers. The company *Atlassian* offered a free *Open Source* license for all three products to the Open Inspire project. Several tools such as *Bugzilla*, *Trac* and *Redmine* have been evaluated before and used with OI until 2010 but have been replaced by the *Atlassion* tools due their wider function set and better integrateability.

The selection of tools make it possible to continue and control the OI project even with large groups of developers and users [☒ 3.7.5]. If the project grows, the software *GreenHopper* [gre] provides all services for an agile project management, including backlogs, sprint planning and project tracking[68] Beyond that it is possible to introduce automatic tests and notifications about build problems using *Jenkins*, which also integrates well into the *JIRA* environment.[69]

4.6. The Build Environment

A build system usually comprises one or more scripts for the compilation of source files, the preparation of resources and the subsequent composition of all build artifacts to an application

[68] GreenHopper is integrated into the OI environment, but disabled until an agile development will be started.
[69] It is for example possible to link *JIRA* issues to *Jenkins* builds.

package. The aim is to automate as many steps as possible while minimizing interactions with the developer during the build process. The build system thus has a direct influence on the development speed since developers need to wait for the time required by the build process, every time changes to the source code or resources are made. To reduce build times, sophisticated systems ensure that only changed files are rebuild and that files, which are not required for the specific target environment, are excluded from the build process. Beyond that there is a strong aim to provide the means for an individual configuration of the build process without compromising the usability. This can be achieved by a simple and intuitive interface to the build system that hides all complex processes, which are executed in the background.

Open Inspire is a dynamic system of modular, loosely coupled and distributed components. This leads to the problem that OI can not be built with traditional static build scripts. Normally each arbitrary configuration of OIMs and OILs would require a specific script that respects the dependencies between modules and the correct build order. Until 2007 OI thus required each OIM to be built individually and all dependencies manually scrutinised. Since the compiler is not able to recognize the assignment of a class to an OIM it is impossible to output a clear message that indicates the missing OIMs. This makes it hard and time-consuming to discover the correct build order. It should be remarked that changing the source code of only one module or library requires rebuilding all modules that depend on it.

With the intention of improving the handling of OI and increasing its acceptance, versions of OI from 2009 onwards come with a more flexible build system, which solves the problems discussed above. The basis of the build system is *Apache Ant*[70][apa11a, HL04], which has been extended through the addition of functionality that supports the dynamic build requirements of OI. In contrast to many other solutions, there is no restriction to a particular OS as well as no requirements for installed programs or libraries other than *Java* and *Ant*.

[70] A considered alternative to *Ant* is *Maven*[apa11b], which already provides some required extra functionality but at the same time a new layer of complexity that comparably need to be adopted to OI.

4.6.1. OI Building Basics

Before covering the internals of the build system, this section briefly explains how to build Open Inspire in just a few steps from the sources. The only prerequisites are a recent Java and Ant[71] installation and the corresponding entries in the default path of the OS. All complex processes are hidden and run in the background so that users without build experience are able to build and run OI. The commands in the following sections do not differ, no matter whether they are executed on *Windows*, *UNIX*, *Linux* or *Mac OSX*.

Checking out the Sources

The Open Inspire sources can be downloaded from the *resources* section of the OI website or directly checked-out from the OI *Mercurial* repository. For developers and everyone who wants to keep the locally downloaded sources up to date, it is a good choice to check-out the sources from the repository. When mercurial[72] is installed and added to the OS's default path, creating a local clone of the repository requires nothing more than the execution of the command:

```
hg clone http://hg.openinspire.org oi
```

The content of the OI repository can be thereafter found in the newly created *oi*-folder.

```
one stefan # hg clone http://hg.openinspire.org oi
requesting all changes
adding changesets
adding manifests
adding file changes
added 321 changesets with 4955 changes to 2008 files (+1 heads)
updating to branch default
489 files updated, 0 files merged, 0 files removed, 0 files unresolved
one stefan #
```

Figure 4.39.: *Getting the OI Sources from the OI Mercurial Repository*

[71] For Open Inspire 1.3 the required *Apache Ant* version is 1.7 or higher in combination with JDK 1.6
[72] Mercurial can be downloaded from the Mercurial project website at `http://mercurial.selenic.com`

Configuring the Build

Before building OI for the first time, a one-off adjustment of the build system needs to be performed. This automatic preconfiguration, in relation to the build environment and target platform, is done by running the following command inside the *system*-folder of the checked-out data:

```
ant configure
```

The build system detects the architecture and build platform, downloads required OIMs and OILs from the OI-LAMP, unpacks and installs them. This is necessary, because the OI repository does not contain any binary files to save storage space and to reduce the required bandwidth[73]. In addition, it ensures that only files, which are relevant for the build process and platform are downloaded. To achieve file integrity, accidentally deleted files are detected and automatically restored from the OI-LAMP.

Directly after the autonomous configuration, a second interactive part asks for some user preferences. The first is the builder's name and email address, which will be integrated into the build and can be queried using the shell command `version`. This allows distinguishing official from user builds and contacting the builder in the case of problems or inquiries. The second question relates to the generation of project files for the integration of OI in Integrated Development Environments (IDEs). Affirming the question generates project files that allow opening OI and components using the *NetBeans* IDE[74]. Finally the JDK used for the build can be selected, if OI should not be build with the system wide standard *Java* installation.

Building the OI Release

After the build system has been configured, it is possible to build OI using the command:

```
ant build
```

The system builds the OI Kernel, all system libraries, required OIMs, downloads further dependencies from the OI-LAMP and creates the release folder, introduced in section 4.1.5

[73] VCSs save space by storing only the file changes between consecutive check-ins. This however only works for text files and not for binary files that occupy much space since all versions of the same file are still present.
[74] More details about the *NetBeans* integration can be found in section 4.6.4 on page 170

on page 83. The OI modules to be built, can thereby be set through a configuration file, which will be covered in section 4.6.2 on page 166. All required dependencies will be automatically resolved and the build order will be calculated by the build system. Whenever `ant build` is called again, only changed files will be compiled and copied. No user intervention is required.

Cleaning the OI Release

If the build system needs to be reset to its original state and all built binaries and created release files deleted, this can be done using:

```
ant clean
```

The configuration files and downloaded data remains untouched in order to speed up later builds. If it is required to reset the system to the state it was in directly after the checkout, this is done by:

```
ant verylean
```

The system hence needs to be reconfigured using `ant configure` prior to the next build.

4.6.2. The Build System Layout

OI developers can be split into the three categories: *Kernel Developers*, *Module Developers* and *Accessory Developers.* All three groups have, contrary to users, an interest in the internals of the build system. While section 4.1.5 covered the *release*-folder, the following pages outline the *system* and *mods*-folder that contain relevant data for kernel and module developers.

Figure 4.40 documents all relevant contents of the *system* folder. The folder's root contains the *Ant* script *build.xml*, which is the entry point for all build targets and delegates the execution to further build scripts in *oiproject* and *prj*. The build system is for clarity reasons structured into directory-based subparts to allow building the kernel or libraries individually.

Documentation - doc

The first place to go after checking-out the sources is the folder *doc*, which contains the files *README.TXT*, *TODO.TXT* and *FAQ.TXT*. The first file contains build and installation instructions, version information and a short introduction. The *TODO.TXT* informs about

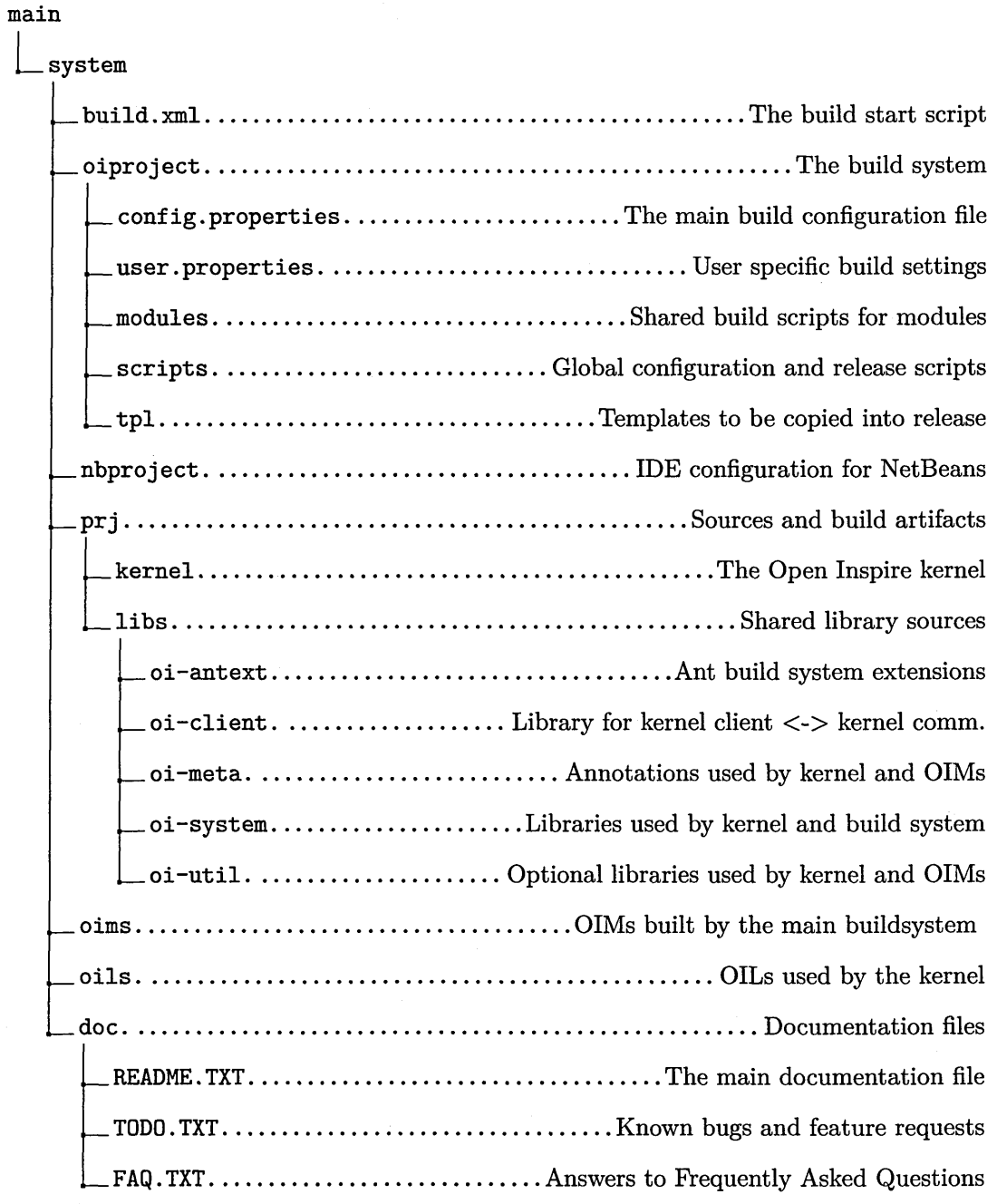


Figure 4.40.: *The Build System Layout*

known problems and the development roadmap and the third file answers frequently asked questions.

OI Kernel - kernel

The most important folder for OI core developers is named *kernel* and is located inside the *prj*-folder. It contains all sources and resources of the OI Kernel. In addition to the sources that can be found in the subdirectory *src*, the kernel's build script *build.xml* is stored here. This script, which is called during the build process, compiles all classes, stores them into the *cls*-folder and uses the created build artifacts to create the packed OI binary *OpenInspire.jar*.

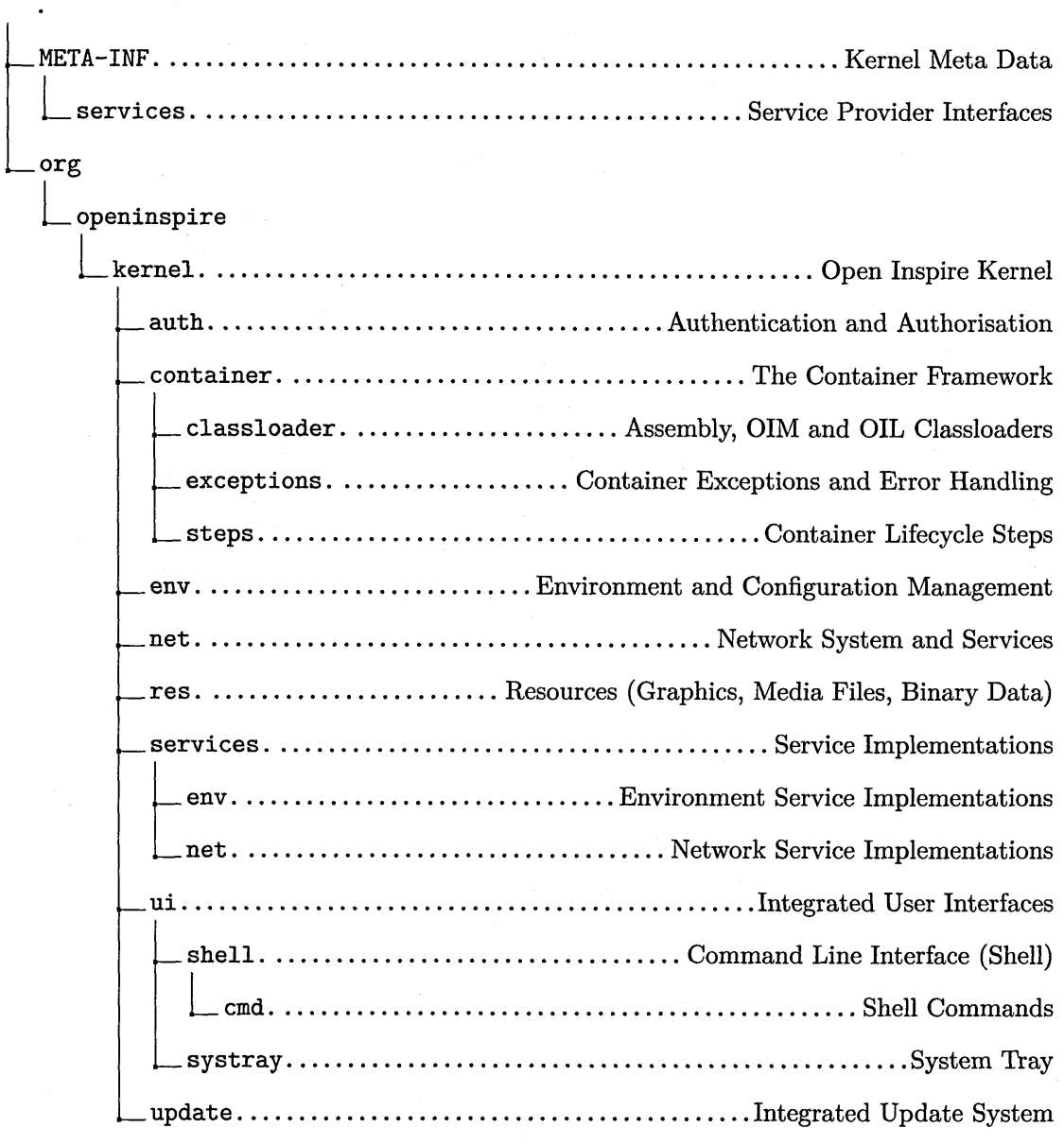


Figure 4.41.: *The Package Hierarchy of the Kernel Sources*

Figure 4.41 shows a rough overview of the kernel's source package structure combined with short descriptions. A detailed explanation of the approximately 100 kernel classes however goes beyond the scope of this written thesis and will be omitted for this reason. Details can be obtained directly from the kernel sources that both come with *Javadoc* comments and more detailed information.

System Libraries - libs

In addition to the kernel sources, OI comes with the five libraries *oi-antext*, *oi-client*, *oi-meta*, *oi-system* and *oi-util*, which can be found in homonymous subdirectories under *prj/libs*. Each library can be built independently, contains its own *build.xml* and *src*-folder, stores compiled data into a *cls*-folder and creates a JAR file.

Instead of integrating all functions directly into the OI Kernel, outsourcing functionality into individual libraries contributes to the reusability and decoupling of the system. External programs normally use only certain subsets of the overall functionality so that splitting the whole function-set into five small libraries instead of one large is meaningful. This does not only save space but also prevents misuse, since external programs can exclusively access the limited function set, provided by the specific library.

Figure 4.42 illustrates the relationship between the libraries and subdomains of the overall system. The division has been deliberately chosen to allow each library acting as a mediator between specific parts of the system. This is necessary to achieve a well defined data-flow between kernel, build system, modules or external clients using provided APIs and entities.

OI Ant Extensions - oi-antext

The library *oi-antext* is the first part that is built by the OI Build System. It contains so called *Ant-tasks* that have been written to extend *Apache Ant* [apa11a, HL04] by dynamic features, required for the complex build process of *OI*. It thus provides functionality for the dynamic handling of class paths, sorting and resolving module dependencies, downloading and installing OIMs and OILs from the OI-LAMP as well as providing the means for the formatted output of text messages during the build process. The speciality of the library is

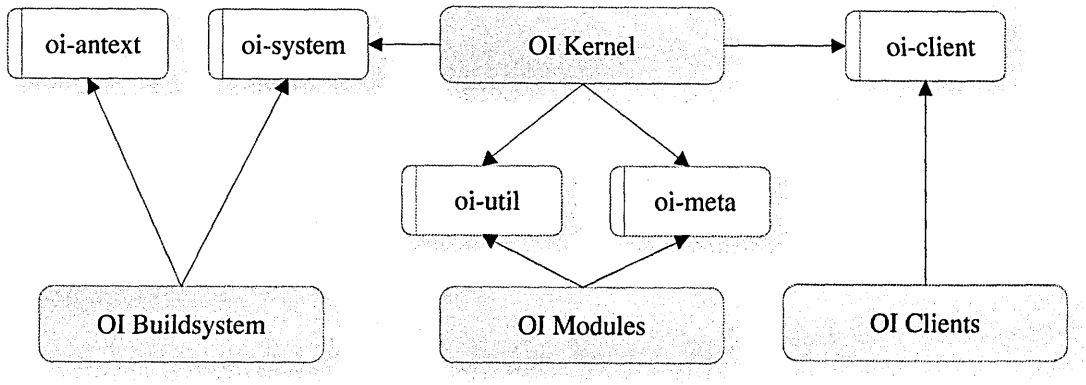


Figure 4.42.: *The Relationship between the OI Libraries and the Overall System*

that it is build and directly used in the same process due to the fact that the further build process itself needs functions of the freshly built library.

```
org.openinspire.lib
    └─antext.....OI Ant Tasks for Buildsystem
      └─ui.....Build System User Interaction Classes
```

Figure 4.43.: *The Hierarchy of the "oi-antext" Library Sources*

OI System Library - oi-system

The second library is called *oi-system* and used by the OI Kernel, build system and OI-LAMP. It includes functionality for the analysis of the execution environment, classes and entities for the management of OIMs and OILs as well as network functionality for the interaction with the OI-LAMP. The primary aim is to provide wrappers and tools that simplify the handling of modules and libraries. A high level API allows the easy installation of modules directly from the OI-LAMP, while low level functionality automates nearly all complex tasks in the background. These tasks include creating and installing components, reading and writing meta-data, calculating and resolving dependencies as well as browsing and downloading from the online repository. An important feature is the automatic detection of the runtime environment and the associated download and installation of platform dependent libraries. Another complex feature is the topological sorting that is used by both the container

and the build system to calculate the build-, injection-, instantiation- and execution order of modules and their dependencies.

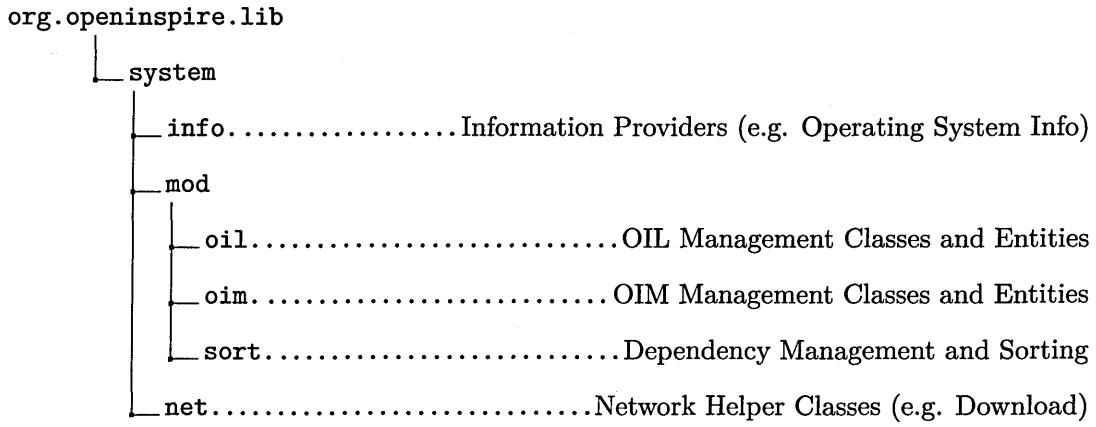


Figure 4.44.: *The Hierarchy of the 'oi-system' Library Sources*

OI Client Library - oi-client

Open Inspire provides a network service[75] that allows clients to remotely access the OI Server, OI Container, running OIMs, Assemblies or stored data. The *oi-client* library in exchange provides an API for all client-side functionality, necessary to create encrypted tunnels, set up connections, authenticate and communicate with OI servers. The server on the other hand uses the library for the registration of services with help of service interfaces, which can be found in the *services-* folder of the library. How these can be accessed using OI's network based *Lookup* mechanism will be covered in section 5.1.1 on page 176.

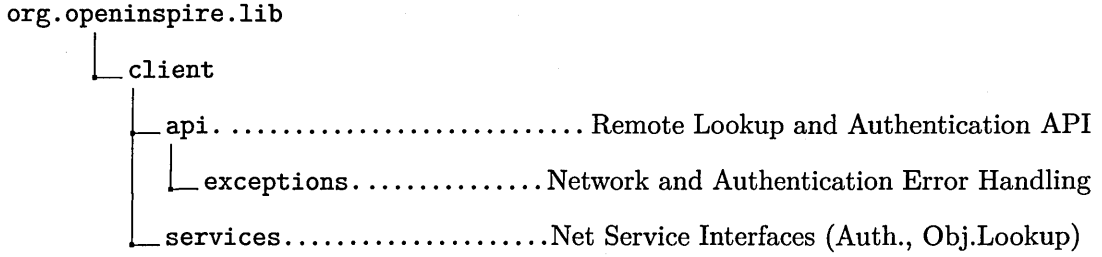


Figure 4.45.: *The Hierarchy of the 'oi-client' Library Sources*

[75] See section 4.4.1 on page 141 for a brief introduction to the network service and page 143 for client usages.

OI Meta Library - oi-meta

The library *oi-meta* is, with only three classes, the smallest library. It contains the two annotations *OIProperty* and *OIPort*, illustrated in figure 4.9 on page 99, and a third called *OIEvent*. This can be used for a dataflow based delegation of events between OIMs, which is however not used in this thesis in favor of the classical events provided by the *Java* API.[76]

```
org.openinspire.lib
    └meta..... OI Meta Classes (OIPort and OIProperty Annotations)
```

Figure 4.46.: *The Hierarchy of the "oi-meta" Library Sources*

OI Utilities Library - oi-util

The last library *oi-util* contains utility classes for the use by OIMs. To make the library settable as module dependency, it is also wrapped by the OIL *oil-oi_1.3.x-x_all*. The sub-packages *entities* and *io* extend the *Java* standard API by new datatypes such as *Pair* and methods for the comfortable copying of files[77]. The *oi-util* library is similarly to the client library equipped with service interfaces, but this time for a local *Lookup*. This allows currently running OIMs to access common container functions such as getting public information of other OIMs inside the container or querying the path where the specific OIM may store data. This is for security reasons, because the read and write access to the filesystem is explicitly restricted to one folder for the specific OIM instance, a second shared folder for all OIMs of the same type and a third for all OIMs. Finally, the last sub-package, *sync*, contains classes that provide mechanisms used to synchronize OIMs. One use-case is the blocking of the execution of a program until a position of an axis is reached.[78]

OI Libraries - oils

Due to space and bandwidth considerations, as discussed in the section "Build System Configuration" on page 157, no binary files are stored inside the OI Mercurial repository. Since OI requires some prebuilt binary libraries for the build process, these are automatically down-

[76] Dataflow based event delegation is planned to be replaced by a dataflow subcontainer architecture
[77] This is necessary, since convenient file handling is first available as part of JSR 203 in the future JDK 7
[78] More details about OIM synchronization can be found in section 5.3.2 on page 197

```
org.openinspire.lib
    └─util
        ├─entities.....Special Datatypes (e.g. Pair)
        ├─io.....IO and Stream Utils (e.g. File Copy)
        ├─services.....Internal OI Service Interfaces
        └─sync.....Extended OI Synchronisation Utilities
```

Figure 4.47.: *The Hierarchy of the "oi-util" Library Sources*

loaded from the OI-LAMP. To realize this with already existing means, the build system comes back to the same OIL design, used inside the container. If a missing library is detected, it is downloaded as OIL from the OI-LAMP, stored inside the *oils/packed* folder and unpacked to *oils/unpacked.* From here it is possible to directly access the individual libraries that were wrapped inside the OILs. Also the previously introduced system libraries inside the *lib*-folder are published using the OI-LAMP. The OIL *oi-kernel* contains the libraries *oi-system.jar* and *oi-client.jar* and the OIL *oil-oi* comprises the *oi-meta.jar* and *oi-util.jar.*

OI Modules - oims

If only the folder *system* is downloaded from the repository, the build system will consequently exclusively build the pure OI Server without additional modules. If a developer also wants to build OIMs, he can either download the additional *mods*-folder, which contains the sources for all official OIMs, or create a *mods*-folder to store own OIM developments. The *mods*-directory is located on the same level as the *system*-folder and not within. This decouples the OI Server from independent OIMs and lets the user decide whether he wants to use only the *system*-folder, the *system*-folder and a selected number of modules or the whole repository.

If a new OIM is developed, the developer can either start from scratch, adapt the template *org-openinspire-oim-template-module* or use the module generator from the OI Website. Each OIM has its dedicated user configurable build file, a *src*-directory for the sources and a configuration file named *module.properties.* This file contains the name, version, description, authors and dependencies to other OIMs and OILs and serves as data source for the OIM

Manifest, covered in chapter 4.2.1 on page 92. If no generator is used, the manual configuration is also simple since all values are self-explanory and sufficiently documented.

For a better overview and handling, all OIMs are separated in groups. An index with all official OIMs and their related groups can be found on the OI website.

Build System Resources - oiproject

All universally reusable resources of the build system are located at a central place inside the *oiproject*-folder. This folder contains the two build configuration files *config.properties* and *user.properties* as well as the three folders *modules*, *scripts* and *tpl* with build scripts, resources and release artifacts.

Build System Configuration

By means of the two build configuration files it is possible to configure the build system flexibly without ever having the need to touch a build script.

The first file *config.properties* contains internal properties for the build process that are never changed by users normally. It contains the current version number of the build, compiler options, paths to sources, and target directories as well as information about build dependencies. Beyond that, it allows the configuring of the release by setting the default paths and filenames, configuration properties such as the URL of the *OI-LAMP* and by selecting the example *Assemblies* and configuration files that are added to the release. This file is normally uninteresting for users and developers and is only changed when a new official version is released. More detailed information can be found in form of comments directly inside the file.

More interesting is the second file *user.properties*, which enables users to customize the build process. The file is automatically generated by the *configure*-task and inter alia serves as storage for the interactively requested configuration values such as the username, contact data, selected build platform and whether *NetBeans* project files should be generated. Beyond that it is possible to select which OIMs or OIM-groups are to be built. This for example

would allow the whole group *calibration* and the individual module *math* to be built, while excluding all other modules found inside the *mods* folder.

4.6.3. The OI Build Targets

Just as important as the layout of the build system is the dynamic behavior of the individual build tasks. This section gives a brief overview of the previously discussed *configure*, *build*, *clean* and *veryclean* targets as well as the yet to be described targets *info*, *run*, *rebuild*, *cis-config* and *profile*.

Each build target can be executed by `ant <targetname>` inside the *system*-folder. The main build script automatically branches into all dependent build files and executes all the relevant sub-tasks of the target. This process is accompanied by informational and warning messages and gives a good overview about the specific build sequences. A detailed description of the build sequences is thus omitted in favor of the following short description of the targets.

The info Target

The *info*-target can be invoked by `ant info` and prints the OI Version, Build ID and a list of all primary build targets including short descriptions.

The configure Target

The *configure*-target starts with cleaning the build system from already existing configurations and obsolete OILs. This step is followed by the download of all OILs from the OI-LAMP required for the further build process. These are inter alia used for the creation of the library *oi-antext*, which is again directly integrated into the system to provide OI's advanced build functionality. This library finally acquires the user values, covered in section 4.6.1 on page 157, which are used as input for the creation of the configuration file *user.properties*.

The build Target

The *build*-target first builds the libraries *oi-system*, *oi-meta*, *oi-client* and *oi-util* as well as *oi-antext* if it is not already available. These libraries are required by the kernel, which is built in the next step and configured with the personal values from the *user.properties*.

Next, all modules inside the *mods*-folder that are also activated in the *user-properties* file are analyzed and evaluated for their dependencies. If dependencies cannot not be resolved locally, they are downloaded from the OI-LAMP and made available to the build system. When all dependencies are resolved, the topological sorting algorithm, provided by the *oi-system*-library, calculates the correct build order before building and installing all selected OIMs. Finally, the release structure that is shown in figure 4.4 on page 84 is created and populated with the kernel, system libraries, OIMs, OILs, configuration files, example assemblies and startup scripts. Depending on the target OS, all files are configured with the OS specific file permissions and the startup scripts and OI main program are made executable.

The clean Target

Calling the *clean*-target causes the execution of the internal *clean*-targets the kernel, all libraries, all OIMs and the release script. This results in the deletion of all built artifacts such as the compiled classes, Java Archives and the complete release folder. It equals a reset of the build system to the state after the execution of `ant configure`.

The veryclean Target

Executing `ant veryclean` includes the target *clean* but also deletes the *user.properties* and any downloaded OIL as well as generated project files for the IDE-integration. It equals a reset to the state after the checkout and requires running `ant configure` again.

The rebuild Target

`ant rebuild` is a convenient method to rebuild the system. It combines the *clean* and *build*-target in one command and rebuilds all files without leaving out already built classes.

The run Target

With help of the *run*-target it is possible to start the OI Server directly from the build system. The command changes to the *release* folder and executes the OI specific startup script.

The cis-config Target

If Open Inspire is built by a Continuous Integration System, this cannot react to the interactive configuration questions of the *configure*-target. The *cis-config*-target thus allows the execution of the configuration step of the build system in a non-interactive mode by passing the user configuration values using a configuration file. This target is never called by users.

The profile Target

The final target *profile* is exclusively used by *NetBeans* IDE, which is covered in the following section. It allows profiling OI to get information about the runtime behavior and execution time of specific system components. Calling this target manually results in an error message.

4.6.4. IDE Integration

Every developer prefers to use an editor and build environment that integrates smoothly into the workflow. The build system thus aims to be independent from specific development tools and is based on standards such as *Apache Ant* that may be directly used from the command line or integrated into Integrated Development Environments (IDEs). If *Java* and *ant* is installed, this is enough to start developing, no matter which Operating System or editor is used.

To speed up the development and improve the software quality it is recommended to make use of an IDE that provides sophisticated software engineering features such as code refactoring, advanced debugging, profiling, auto completion and the automation of recurrent tasks [⊠ 2.3.6]. All these features are provided by the four major and widespread *Java* IDEs *Eclipse* [ecl11], *NetBeans* [net11], *IntelliJ IDEA* [int11] and *JDeveloper* [jde11]. While all IDEs are suitable to be merged with the OI build process, it is beyond the scope of the thesis to provide implementations for all these IDEs. Instead, the decision was made to provide only one reference implementation based on *NetBeans*.

The first *NetBeans* Version was published in 1998, for more than 10 years it has been the IDE which is the longest on the market. It has the second largest community after eclipse and the highest immigration of new users, so that a large number of developers are addressed[79].

Over the last years *Eclipse* had a high popularity due to its snappy user interface built using the proprietary GUI framework SWT, which allows a fast and native rendering using the OS's standard look and feel, while *NetBeans* came with a *Swing*-based UI that did not integrate as well and was noticeably slower. In the meantime *Swing* has been improved and comes with a native rendering so that *NetBeans* has the advantage that it builds on the standard *Java* GUI library, is at least as snappy as *Eclipse* and integrates even better into the native system look and feel.

IntelliJ IDEA is a sophisticated and professional IDE that is published as *ultimate* and *community* edition. The problem is that OI is strictly based on free open source projects, which only makes the community edition of *IntelliJ IDEA* eligible and this is missing some important features. *JetBrains* gives free of charge ultimate edition licenses to open source project developers but it is not possible to prove the entitlement of each volatile OI developer.

JDeveloper is as well as *NetBeans* developed by *Oracle* and hence has the advantage that is comes from the same company as *Java* and so quickly follows new *Java* releases. It provides functionalities for advanced *Java EE* developments but is less extensible than *NetBeans*, which explicitly focuses on providing a modularly extensible platform. The simple usage, central plugin-management, sophisticated UI and the presence of the extensible *NetBeans* RCP are thus key factors for the choice of *NetBeans* as standard IDE for Open Inspire.

NetBeans IDE Integration

To keep OI independent of a specific IDE, the build system is decoupled from the IDE integration. If the optional support for the *NetBeans* IDE is required, it can be activated during the execution of the *configure*-task. The first build thus automatically generates a *nbproject*-folder in the *system*-directory, which creates the required *NetBeans* project infrastructure. In addition to the main project, further project files are created inside the *mods*-folder of each

[79]The analysis has been performed using data of the Google Trends engine [goo11] using combinations of *netbeans,netbeans ide, eclipse, eclipse ide, intellij* and *jdeveloper*, web publications and developer interviews.

activated OIM. From now on, OI automatically updates all IDE project files to reflect the OI project infrastructure with their complex dependency structure. The challenge is here to make the IDE finding all source and resource dependencies stored in OIMs, OIL and system libraries, incorporating OI's access restrictions between components as well as following the component versioning.

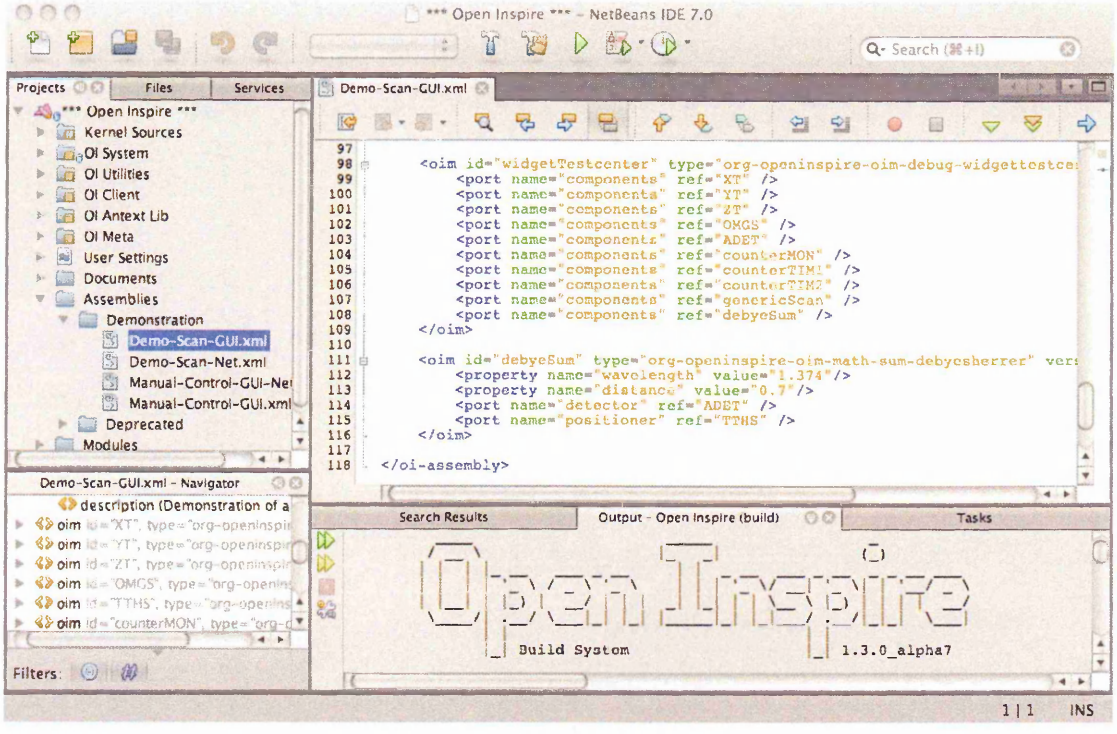


Figure 4.48.: *Open Inspire NetBeans Integration*

If the *NetBeans* support is activated, the *system* folder and all folders of activated projects can be, as any other *NetBeans* project, directly opened with *NetBeans* 6 or higher. This feature works out-of-the-box without additional plugins. Figure 4.48 shows the opened OI main project in *NetBeans* 7.

The Project Explorer

The left side of figure 4.48 shows the *Project Explorer*, which has been customized to reflect OI's project structure. The tree shows nodes to provide direct access to the sources of the kernel and five system libraries *OI System*, *OI Utilities*, *OI Client*, *OI Antext* and *OI Meta*.

The next tree node is a direct link to the *user.properties* and opens the file using the editor. The last three nodes finally provide direct access to the OI documentation, show a tree of the OI *Assemblies*, which are copied to the release, and all OIM sources inside the *mods*-folder.

The Files Explorer

While the *Project Explorer* provides direct access to all frequently used files, the *File Explorer* shows the unfiltered content of the whole *system*-folder. This is useful when files should be browsed or opened that are not accessible by the *Project Explorer*. These include build scripts, internal configuration files, templates or build artifacts.

The Editor

The *NetBeans* editor right beside the *Explorer* allows viewing and editing all text based files that come with Open Inspire. The first apparent feature is the *syntax highlighting* that improves the readability of all important files such as *Java* files, *Assemblies*, configuration files and build scripts by applying syntax specific text coloring and font styles.

Further features that make use of OI's dynamic dependency management are the automatic code completion, just in time compilation for syntax checks and refactoring mechanisms. These mechanisms all require the dynamic resolution of dependencies between loosely coupled OI components, which is provides by OI's dynamic adoption of the *NetBeans* project configuration files.

If an OIM is edited that has dependencies to other OIMs or OILs, this information is also bypassed to the editor. This feature is important for the *code completion* and *auto import* features that belongs to the key mechanisms required to simplify the creation of new modules.

If for instance a new *Counter*-OIM should be created that has a dependency to the module *domain-science*[80], it is possible to automatically create the source skeleton of the new counter. Figure 4.49 shows that it is only necessary to add `implements Counter` to the new counter class called *Example*. *NetBeans* then proposes to implement all abstract methods, which results in the automatic import and creation of all method bodies of a standard OI counter

[80]The Interface Domain *org-openinspire-domain-science* has been introduced in section 4.3.3 on page 125

module. Beside this auto creation it is also possible to start writing variable or class names and let *NetBeans* complete it according to already defined specifiers that fit in the context. More information about *auto completion features* such as the encapsulation of fields in getter and setter methods can be found in the manuals on the *NetBeans* project webpage [net11].

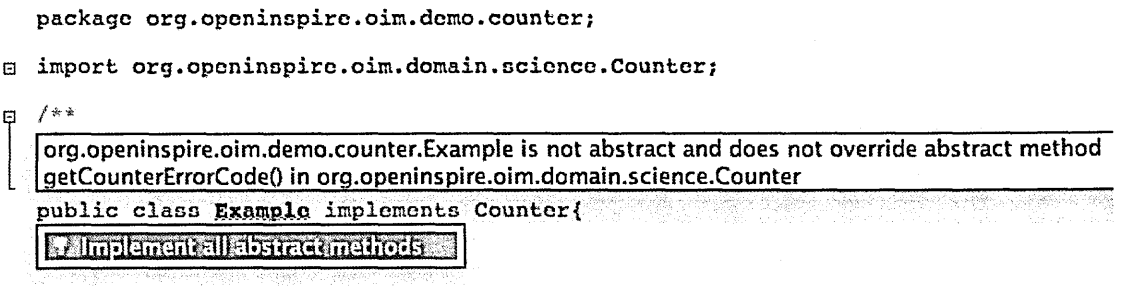


Figure 4.49.: *Auto Completion in NetBeans Editor*

Another related mechanism to the *auto completion* is the *refactoring* support that allows quick and parallel modifications of source code in multiple files and modules. This makes it possible to change the name of a package, class, method or variable at one place and let the IDE automatically adjust all sources to use the new name. This also works for method parameters, allows moving method and classes and provides a way to securely delete variables, methods, classes and packages by checking all dependencies in all relates loosely coupled OIMs.

Toolbar and Menus

The toolbars and menus has been adjusted to support the targets provided by the OI Build System. This means that executing *Clean*, *Build*, *Clean and Build*, *Run*, *Debug* and *Profile* from the *Toolbar* or from the *Run*, *Debug* and *Profile* menu triggers the corresponding targets provided by the OI build system. All other OI specific build targets (see page 167) can be started using the context menu by right clicking the main project in the *NetBeans Explorer*.

Output Window

If an ant target of the OI Build System is executed, all messages are redirected to the output window at the bottom of figure 4.48 and provides messages about the build process in black and warnings in red. Also the OI *Shell* is executed in this area of the *IDE* window.

Debugging and Profiling

Apart from the manually executable tasks from page 167 exist the two targets *Debug* and *Profile*. The debugger assists finding bugs through the help of features such as the stepwise execution, setting of breakpoints and creating watches that display runtime execution. The profiler on the contrary measures execution times and memory usages of selected program fragments. It can be used to analyze the runtime behavior to ease fixing performance problems and revealing memory leaks.

Version Control

The last notable feature is the seamless integration of the version control system *Mercurial*. It inter alia allows checking out OI directly from the *IDE*, visualizes code changes in a graphical way, allows the interactive merging of conflicting files[81] as well as checking-in the changes.

More detailed information about features that are not directly related to the OI Build system can be obtained from the *NetBeans* website [net11]. Examples are the visualization and generation of UML diagrams or the GUI builder *Matisse*, used to create graphical OI Widgets.

[81] This is important when multiple users change the same piece of code.

CHAPTER

FIVE

OPEN INSPIRE IN THE FIELD

The chapter Open Inspire in the Field covers concrete realizations of projects built on and with Open Inspire. Importance is attached to present a wide-ranging selection of different types of examples in the area of hardware control, simulation, data storage, scans and user interface clients. Bridging to the legacy control system Caress and accessing instrument hardware will be covered as well as *Monte Carlo* [Bin79] based experiment simulations with McStas, experiment data storage using the NeXus format, simple and advanced diffraction scans with aid of the Inspire Scan API and the development of Graphical User Interfaces such as a Rich Client built on NetBeans RCP.

5.1. User Interface Examples

Open Inspire is specifically designed to be independent from a fixed User Interface. The OI server and therein executed modules provide services and these can be visualized or controlled using so-called *thin-clients*. *Thin-clients* themselves contain only the absolutely necessary logic required to visualize data and to give users the ability to control the OI server remotely.

This paradigm makes it possible to develop independent clients that all access and control the same functionality on the server but can be tailored for specific OSs, environments or

users. The development of new forms of User Interfaces (UIs) emerges fast. Years ago, the standard way to interface a system was the *command line*, the last years were marked by *mouse-controlled* user interfaces and now multi-touch based UIs, as found on mobile phones and tablets, become increasingly popular.

The proceeding technology development on the client side is much faster than on the server side, which makes it reasonable to decouple the development on both sides. The server side functionality hence needs to be developed only once and be kept untouched when new types of clients arise.

5.1.1. Client Binding

To achieve a decoupling between server and client it is expedient that the data and control interfaces provided by the server are in a general format rather than aiming at a specific client to be interfaced. A good solution is to use a middleware that allows server interfaces for all provided functions to be defined and to create client interfaces at a later point. Solutions such as CORBA [OMG04] or *ZeroC ICE* [Hen04] for example provide programming language independent interface definitions that can be used to create all necessary code for different programming languages and hence make it unnecessary that clients are written in the same language as the server.

This solution however is not sufficient when it comes to the dynamical features of the OI *Container* and does not meet all requirements. When programming language specific code is generated from an interface definition, this code is static and can not reflect changing interfaces on the server side. Yet unknown OIMs can be loaded on demand and even coexist in different versions with different interfaces in the same container. A client hence must be able to recognize which interfaces are provided by the server and dynamically supply client-side functionality for yet unknown services. Beyond that, being bound to a fixed middleware solution prevents the system to be adopted to upcoming new standards and developments.

Remote Lookup

OI loosely couples the client and server using a *Lookup*-Mechanism and introduces a wrapper that allows the changing of the underlying network / middleware solution without needing to

reimplement any client or server parts. The core idea of the concept is to provide a server-side object registry[1] that is used to register *plain objects*[2] and make them available to be accessed locally and by remote clients. *Plain objects* means that the classes of the registered objects do not have any relation to OI and consequently do not need to implement any interfaces or to extend classes.

The registry is realized using the server-side class `Object Registry` and implements the interfaces `ObjectRegistryLookup` and `ObjectRegistryService`. While the second interface is only accessible by the server and OIMs, the `ObjectRegistryLookup` is stored inside the independent library *oi-client*, which is also used by the client. It provides all methods that are required to access and control remote objects, handle events and notifications between objects, provide authentication mechanisms and query interface and class inheritance hierarchies. The `ObjectRegistryService` on the other hand is only available on the server-side and allows the registering and unregistering of objects and the setting of access privileges. These interfaces act as a facade and are independent from the underlying network implementation. Switching the network protocol or middleware solution hence only requires the replacement of the *oi-client*-library by one that implements the new protocol, while the client and server code remains untouched.

The Server Side

Two different types of services are published through the registry: Services provided by OIMs inside the container and the *Shared Services* introduced in section 4.4.1 on page 140, which give access to global services such as the *Container Lifecycle Management*, *Authentication* and *Filesystem*. While all shared system services are registered automatically at startup by adding the objects to the registry, the registration of OIM-provided services can be done in two different ways:

The first way is realized by the OIM `org-openinspire-oim-services-registry`. This OIM has an *OIPort* called *services* that allows to connect an arbitrary number of OIMs to it and

[1]The core of the object registry is a simple HashMap that holds references of objects and allows to quickly get an object for a given key.

[2]Currently only *Java Objects*, because the *OI* Server is written in Java. But the design is flexible enough to register *SCALA*, *Groovy*, *Jython* and objects written in other languages that run inside the JVM.

automatically registers the OIMs service interfaces to make them publicly available to clients. One advantage of the concept is that the OIMs are not bound to the server but only to the *Registry* OIM, which can be replaced in a simple way. Another advantage is that the OIMs can be selected that should be published and all OIMs that are not connected to the registry OIM remain invisible to clients. To publish the service interfaces of all currently loaded OIMs, it is of course allowed to add a registry-OIM with the wildcard (*) in its service port:

```
1 <oim id="serviceRegistry"
     type="org-openinspire-oim-services-registry"
     version="1.3.0_alpha">
2        <port name="services" ref="*"/>
3 </oim>
```

Figure 5.1.: *Register the service objects of all active OIMs using a wildcard reference*

The second way, beside the OIM-based registration, can be gone when more than one service interface or another interface than the service interface of the OIM is to be registered. Here it is possible to register objects simply through the `register` and `unregister` methods of the `ObjectRegistryService` interface. While the `ObjectRegistryLookup` has been stored into the *oi-client*-library to be accessible by clients, the `ObjectRegistryService` is as well available through an external library to remain the OIMs independency from the OI *Server*. This library is called *oi-util* and contains utility classes usable by OIMs, the server and clients.

The Client Side

A look at the client-side shows the simplicity and flexibility of the approach. The reference implementation for the remote lookup mechanism provided by the *oi-client* library uses the RMI protocol to access the server-side object registry. To get access to the registry from the client, only the following line of code is required:

```
RemoteObjectLookup lookup = new RemoteObjectLookup("servername", 2008, 2009);
```

The command requires the network name of the OI server, the RMI registry port and a second communication port. The ports both use TCP and default to 2008 and 2009 for Open Inspire. No other ports are required so that it is very easy to tunnel all traffic through

firewalls and to access the instruments at the HZB (for instance using an SSH tunnel) [☒ 5.3.5] [☒ 6.5.2].

Now it is possible to run several commands on the returned object *lookup*:

- `List<Object> lookupAll()` is the simplest command and returns a list of all publicly registered remote-objects, which can now directly be accessed from the client.
- `<T extends Object> List<T> lookupAll(Class<T> interfaceClass)` is much more sophisticated, because it allows the return of only the objects that implement a specific interface. The underlying functionality is executed on the server and recursively searches if the given interface-class appears in the inheritance hierarchy of the published object.
- `<T extends Object> List<T> lookupAll(Class<T> interfaceClass)` is similar to the recent command but only returns the first object that matches the interfaceClass.

Objects can always be added or removed from the lookup at runtime. The start of an assembly for example registers all selected service objects using the `org-openinspire-oim-services-registry` OIM. Stopping the assembly automatically unregisters the objects. To be informed about these registry changes, it is possible to add a *PropertyChangeListener* that is always called when the content of the lookup changes. OI comes with standard *Java* property change mechanisms, which can be called on the *lookup* object:

- `addPropertyChangeListener(PropertyChangeListener listener)` to add a new *PropertyChangeListener*, which is called when the *Lookup* changes and
- `removePropertyChangeListener(PropertyChangeListener listener)` to remove the *PropertyChangeListener*.

Authentication

To this point, nothing has been said about security. A client can connect to the OI server without having to pass a username or password and gets a *Lookup* with access to *Service Objects*. If this list of service objects contains all services published by the server it severely breaks the security because everyone could access and even manipulate services [☒ 6.1.2][☒

6.1.4][☒ 3.2.4][☒ 3.2.4]. This problem is solved by a security layer, which assures that restricted services are only published to authorized clients. All other services remain invisible to the client and cannot be forced to be visible since they are filtered on the server-side.

The authentication takes place by calling the method `login` on the already known `lookup`-object and passing a *username* and *password* as parameter:

```
RemoteObjectLookup lookup = new RemoteObjectLookup("servername", 2008, 2009);
lookup.login("admin", "oiadmin".toCharArray());
```

After transferring the encrypted user credentials [☒ 6.3.2] to the server, they are checked by the *AuthenticationService* (section 4.4.1 on page 140). In return all services that are released for the specific *user* and *role* are added to the lookup and the client will be notified about the changed lookup and update the User Interface.

On the server side it is possible to restrict a service to a specific role in two ways: The first way is to use the role-based security mechanisms (section 4.3.2 on page 120) inside the *Assembly* to define the role under which an OIM's service class is published .

The second mechanism is to define that a service needs authentication when a service is manually registered using the `register`-method of the `ObjectRegistryService`. The first method overwrites the behavior of the second method.

It is always possible to change the user and consequently the published services by calling `lookup.logout();` and login with a different username and password.

5.1.2. User Interface Types

User Interfaces can have completely different characteristics. This section shows UI examples with different function sets or different focuses on the platform. The examples help to make it clear how the *Lookup* mechanism works and show the strengths of the approach.

The development of clients is decoupled from the server-side developments so that it is open for everyone to write their own client implementations. This results in a more frequent development on the client-side than on the server-side where normally only bug fixes and

optimizations are applied. To prevent the introduction of clients that maybe come with new or revised User Interfaces (UIs) and are thus outdated when this thesis is published, only simplified examples are presented that help understanding the main concepts. These concepts are usually the basis for all of the different types of clients.

Ready-to-use client implementations are downloadable from the OI website. Screenshots of clients can be found at `http://www.openinspire.org/documentation/screenshots.html`.

OI Shell

One of the simplest client applications is a UI in the form of a Command Line Interface. The difference to the OI *Shell*, covered in section 4.4.2 on page 143, is that the client is not interwoven with the server but connects remotely. It is thus decoupled from the server, and so can be accessed from different computers, allowing multiple users to connect simultaneously and can be implemented using a different programming language than the server. This is meaningful when the shell is directly integrated with a system shell such as the *bash* [Ale01] on *Linux*.

Writing a CLI client is easy due to the lack of a GUI. The first step is to connect to the server and login as explained earlier in this section. The next step is the registration for *Lookup* changes so that the client gets notified whenever a service object is added to or removed from the `RemoteObjectLookup`. Now the client has a list of objects but does not know what to do with them, because the classes or interfaces that describe the functionality are still missing. For clients written in *Java*[3] it is now possible to add an *Interface Domain* (see 4.3.3 on page 123) as library. In this case one with interfaces for all scientific devices, such as provided by the *OIM Domain Science*. A developer can now implement the interfaces step by step, and add functionality for the objects made available by the *Lookup*.

Example: To allow users getting a list of all active *Positioners* via CLI, the *Positioner* interface from the *OIM Domain Science* could be implemented so that a command such as `positioner list` would output a list of all *Positioners* that are currently registered. A

[3]For other languages it is necessary to create the classes using the selected middleware solution.

second command such as `positioner xT get value` could then be used to print the position of the *Positioner xT*.

The essence of the concept is that the available commands vary depending on the currently loaded *OI Assembly* and that it is possible to implement only a subset of all available interfaces. If the *Lookup* changes, the client automatically compares the interfaces of published remote objects with available implementations of the interfaces on the client side. If an interface implementation matches the interface of a published remote object, new functionality that is able to handle the remote object is automatically added to the UI.

Particularly novel is that the system is able to handle interface inheritances. When the client does not find an implementation for a *Positioner* it can query the server for the next matching object in the inheritance hierarchy. According to figure 4.20 an page 126 this is *Device*. When the client has an implementation for *Device* it is now possible to get the value of a *Positioner* but also a *Counter* or *Detector* using a command such as `device xT get value`[4].

Another interesting point is the way that the system handles versioning issues. OI is able to start *Assemblies* that load multiple OIMs with different versions at the same time. The result is that objects with the same interface but a different version can be added to the *Lookup*. Here the design provides means to differentiate between different versions. A client can thus have different versions of the *OIM Science* loaded at the same time and provide the correct version of a UI implementation of an object. This design guarantees that *Assemblies* with legacy OIMs are still usable with clients that provide support for newer versions of the OIMs.

Beyond that it is important to know that the access to remote objects works fully transparently. This means that an object can be called in nearly the same way as if it were local. Even property change mechanisms work correctly, due to the sophisticated handling of the standard *Java* event mechanisms[5]. It is however important to keep in mind that the objects are called via ethernet and connection problems or latencies can occur that a developer must handle by mandatorily catching a set of exceptions.

[4]This is possible since *Device* inherits all methods from *Value*, which provides access to a double value such as the position or counts.
[5]Open Inspire creates hooks that catch *Property Changes* and forwards them to the remote system.

Tailored standalone Applications

A slightly more complex, but also more common client type, is a client formed as a Graphical User Interface (GUI) which uses controls and widgets to interact with remote objects. While the connection to the server and authentication is similar to the CLI client, the way how *Lookup* changes dynamically affects the GUI elements, as will be explained using a simple example.

When an object is added to the `RemoteObjectLookup` it can be made available to a GUI in various ways: It can be added as a desktop widget, integrated as a part of a set of windows, realized as a standalone window on a desktop pane or it can provide its controls and visualization elements in the form of a popup-window. Not all elements shall immediately be visible so that they can be made available using menu items, shortcuts, tree entries or popup-menus.

Figure 5.2 shows the example of a GUI designed in the *Windows Ribbon* Look and Feel, which will serve as a good example to demonstrate the unconventional, as well as the common places, where remote objects can be made available by the UI.

When a client is started but no *Assembly* is loaded by the OI *Container*, only limited functionality is made available through the UI. One example is the container status that can be shown through an information bar at the bottom of the window ❻. An other example is the access to the *Filesystem* through the global *FileSystemService*, which could be visualized by a *Tree View*. Figure 5.2 uses a tree ❸ to provide an ordered list of all currently running OIMs. When the container stops, the items are removed from the *Lookup* and immediately disappear until new items are added. The same mechanism applies for the bar at the top of the window ❷, which provides icons for all currently loaded service objects. These can be clicked and provide the OIM specific functionality in the center of the window ❹. Window ❺ provides access to the global logging service. Here all status messages of the server, container and running OIMs are listed.

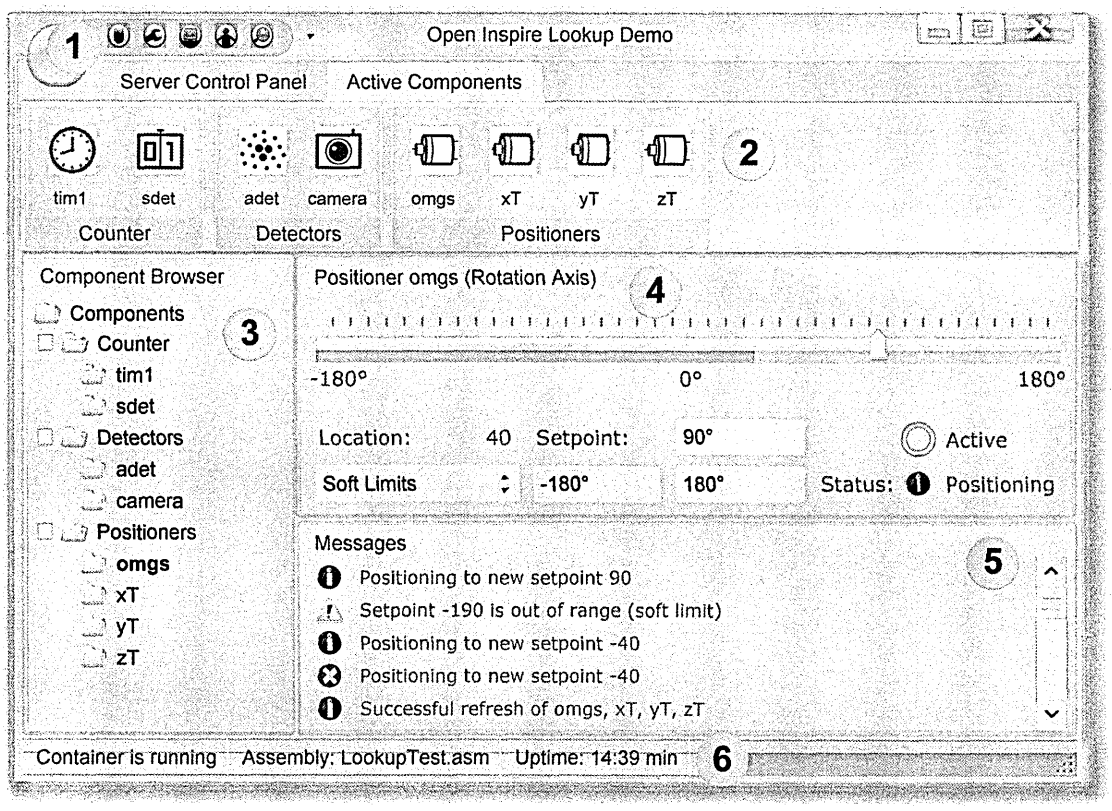


Figure 5.2.: *Specification of the Open Inspire Lookup Browser*

It is important to know that the view differs between users with different roles because the *Lookup* contains only the services that the user is privileged for. To allow an administrator to get the same view as a less privileged user he can switch his role using menu ❶.

One real example of a GUI client is the *OIC Simple Client* that has been developed for the calibration, covered in section 5.6 on page 244. It is similar to the GUI of the *WidgetTestcenter* (figure 4.17 on page 119) but comes with menus that allow to select the windows, which shall provide access to the service objects.

A second important client is realized using the *NetBeans* Rich Client Platform. It is called OI *Sunrise* and developed as a reference for OI clients. *NetBeans* itself already comes with a *Lookup* mechanism similar to OI with the difference that it is only used locally. Developers can put XML files into local folders that all represent specific locations of the GUI and thus trigger the UI to show a menu, open a window or add any other element dynamically to the

view. The *NetBeans Lookup API* that is responsible for this behavior has been extended to support OI's *Remote Lookup* so that *NetBeans* is able to respond to OI's *RemoteLookup* changes in the same way as if local XML files were added. This behavior makes it very flexible and easy to write UI extensions since it is possible to fully rely on the *NetBeans* infrastructure.

When *NetBeans* starts with the *NetBeans* OI extension, it opens a dialog that allows connecting to an OI server after passing a username and password and to optionally connect through an SSH tunnel when an encrypted connection is desired[6]. An important advantage of a Rich Client Platform (RCP) is that it is possible to directly use a large number of advanced APIs. Especially the OI *Filesystem Service* (section 4.4 on page 139) benefits from this out-of-the box feature. If a user is privileged, he can for instance directly access the OI Filesystem through the *NetBeans File Explorer* and has access to functions that allows the renaming, copying, moving, creating or the editing of *Assemblies* and configuration files using the *NetBeans* editor. This editor in turn provides sophisticated features such as syntax checks or code completion since OI provides XML-schemas for all OI XML files such as *Assemblies*.

A User Interface for the OI Store

An other interesting client is a UI that provides a fronted for the OI *Store* (see paragraph 4.2.3 on page 106 and 4.5.2 on page 150). Figure 5.3 shows such an example, which allows to conveniently browse the archive of all publicly available OIMs and to directly install an OIM and all dependencies to an OI server.

The bar at the top allows the selection of an OIM category such as *Science*, *Smart Home* or *Industry*, to sort the OIMs *by type* or by *category*[7], to filter specific OIMs and to search for OIMs. The left side shows all the categories of all available modules that match the search and filter settings. When a category is selected, the modules in this category are listed on the right side and clicking a module shows all information about OIM. This information comes from the OIM manifest and provides links to dependent modules and libraries, a description, a unique name but also a way to select the version of the OIM. If the client also connects

[6]The feature to use an SSH tunnel is directly provided by the OI *NetBeans* extension
[7]This allows to select e.g if a selection shall show only *Positioners* but for all control systems or if all devices shall be shown only for one control system.

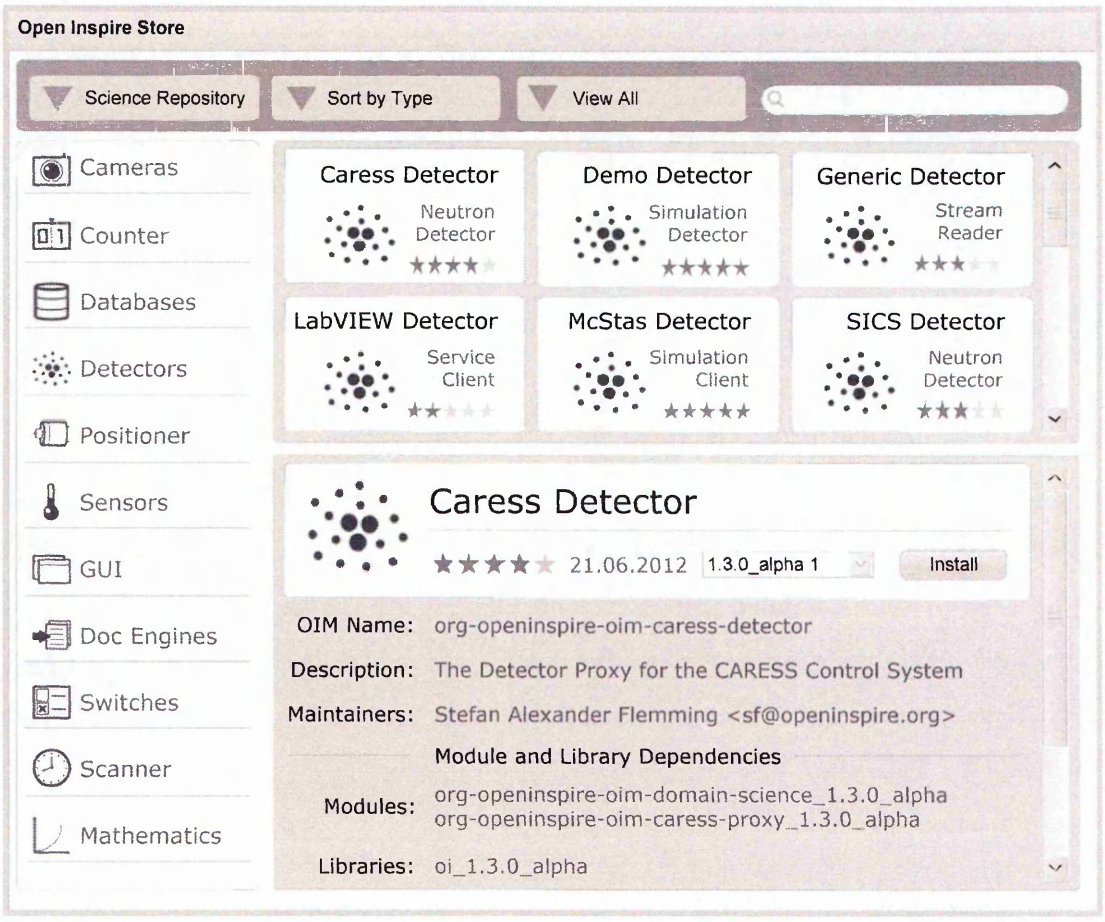


Figure 5.3.: *Specification for the Open Inspire Store*

to an OIM server, it is possible to show which modules are already installed and to directly install modules from the OI *Store* to the server.

The system is similar to the program installation on many *Linux* distributions or online stores for apps such as the *Apple App Store*[8]. The difference is that no standalone applications are distributed but modules that can be connected to form *Assemblies*.

Assembly Editor

Editing *Assemblies* directly in text-mode is error-prone and inconvenient because only advanced editors such as the one used in NetBeans are capable to check the syntax and correct

[8]The first presentation of the OI store was in February 2007, the *Apple App Store* has been introduced in March 2008, the *Android Market* (predecessor of *Google Play*) has been first advertised in August 2008.

hierarchy of XML tags. This is why OI *Assemblies* are designed to be edited using a graphical editor rather than struggling directly with *XML*-files. Using a graphical editor avoids the need to understand the *Assembly* syntax, comes with no function loss in regard to a manual editing and prevents the creation of invalid files due to wiring faults.

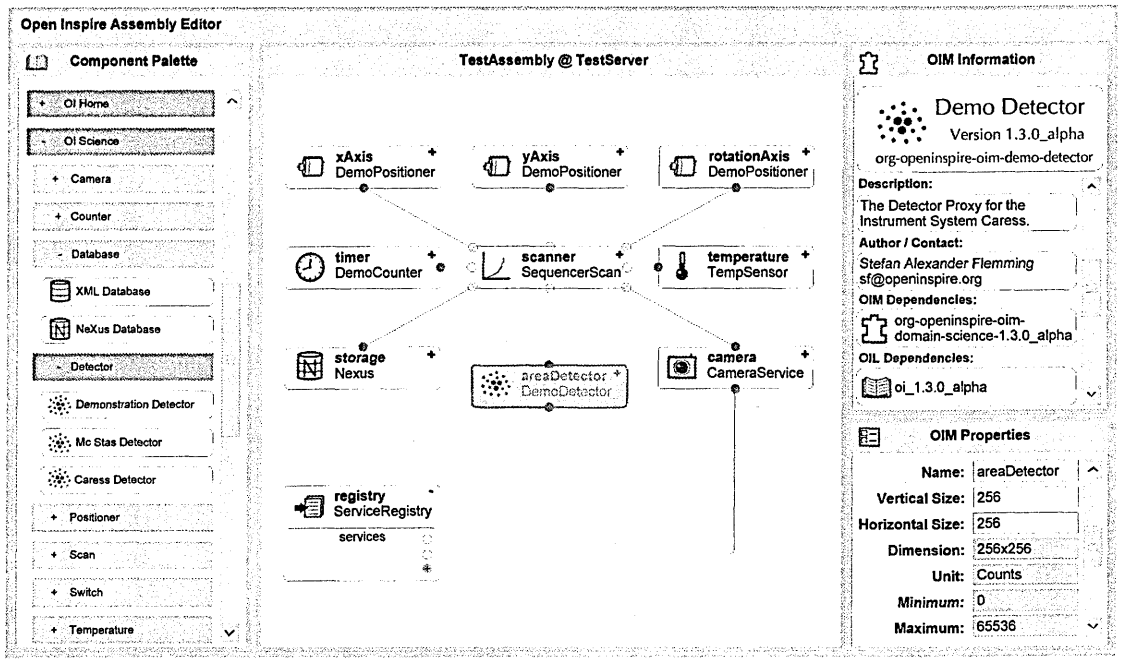


Figure 5.4.: *Specification for the Open Inspire Assembly Editor*

Figure 5.4 shows how a graphical *Assembly Editor* can look like. It allows the editing of a standalone *Assembly* file or connects to an OI Server to edit *Assemblies* remotely. When an *Assembly* is edited locally, it is necessary that the editor has local copies of all used OIMs to read-out the meta Information such as name, description and dependencies as well as the *OIPorts* and *OIProperties*. Here the library *oi-system* comes in, which provides all necessary functions to handle OIMs. This also includes the automated resolution and download of OIM dependencies on client side.

The left side shows a sorted and categorized list of all installed OIMs. These can be dropped to the workspace in the middle. When an OIM is dropped, it is possible to show its *OI Ports* using a small + symbol on the top right. Clicking a port spans a wire that can be connected to an other OIM. This mechanism is fail-safe because it allows connecting ports

only to compatible OIMs. While spanning a wire, all OIMs are automatically highlighted. If only one OIM matches the port, it is connected automatically. When an OIM has a port that allows multiple connections, a new port symbol is added each time a port has been used.

To set the properties of an OIM, it is possible to enter them in the properties window on the right side. The top right sub window shows information about the OIM while the bottom window is used to set the properties, according to the *OIProperties* annotation. A validation of the entered values is automatically performed by means of the *OIProperty*' s datatype.

A proof-of-concept *Assembly Editor* has been integrated into *OI Sunrise* and uses the *NetBeans Graph Library* for the visualization and wiring.

Graphical User Interface Editor

With the standalone client earlier in this chapter, a client has been presented that can flexibly react to a changing environment. To extend or modify the client however, it is required that a developer adds or changes controls and visualizations programmatically to make them available to the *Lookup*. To give a user without programming knowledge the ability to create his own UI, a graphical editor can be used. This is especially useful for monitoring purposes. Figure 5.5 shows a UI *Editor* that demonstrates how to create a monitor client, which could for instance run on a screen directly at the instrument and provide a view with the current state of the instrument.

The editor can be connected to the server and gets a list of all included OIMs. These are shown on the right side. Now the user can drag one of the OIMs from the palette ❶ to the canvas ❸ on the left side. When more than one visual representation for an OIM is available, a window ❷ pops up, in which a widget can be selected. Due to the inheritance hierarchy of the *Interface Domain* (see 4.3.3 on page 123), tailored widgets can be selected that mirror the full functionality of an OIM or any other widget in the hierarchy that consequently comes only with a function subset. A *Positioner* for instance can be represented by 1) a widget that allows the setting of the minimum, maximum and the setpoint or 2) a label that only shows the current position. Such a label is available for all devices that inherit the *Value* interface. To give a user more options to design the UI, it is possible to add a backdrop or graphics.

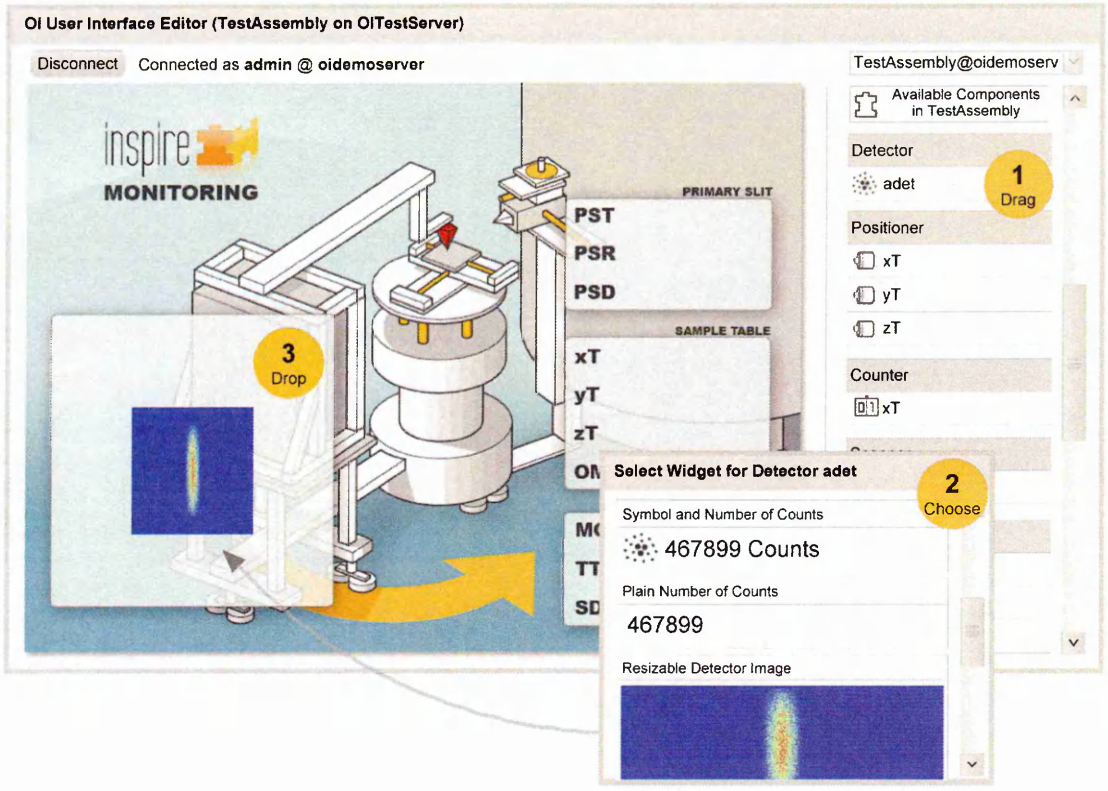


Figure 5.5.: *Specification for the Open Inspire User Interface Editor*

When the design is saved, an XML-file is created with all positions, types and settings of the widgets and the mapping to the OIMs as well as the graphics. Each *Assembly* has a unique id, so that a designed UI can be bound to a specific *Assembly*. This allows the client to view the correct UI when a new *Assembly* is started.

Alternative OI Clients

The earlier examples were all designed as desktop applications, because this is currently the most common way to provide a Human Machine Interface (HMI) that runs on all computers. At least as interesting are clients that are designed as web-applications or mobile apps.

A web application runs on an *Application Server* and provides a UI that can be opened with any browser and is usable on computers but also on mobile devices such as smartphones and tablets. The disadvantage here is that it requires considerable effort to write a web client with a comparable reactiveness and interactivity such as a standalone application. The reason is

the stateless *http* protocol, which does not allow clients to receive asynchronous notification events. The advantage is that the visualization is preprocessed on the server side so that only the visual data is transferred that is required for the view. The *Application Server* is usually installed on the same network as the OI *Server* and can for instance transfer detector data with the full bandwidth while the same data is downscaled and rendered to a small image when it is transfered to the client.

Using a mobile smartphone- or tablet-application to provide a GUI is a very promising approach for a HMI. The user is no longer bound to a fixed location of a terminal but has a portable device that he can use directly at the instrument or elsewhere. This is particularly useful when a person works at the instrument mechanics and want to see or control the values of devices without switching between instrument and computer. Mobile devices are generally cumbersome when it comes to the writing of long texts but for the operation of instruments this is seldom necessary, so that a multi-touch UI has advantages compared to classical computers. Touches can so be used to select controls on a screen that shows the survey of an instrument and gestures allow to zoom into detector visualizations or rotate the instrument.

To prove the concept, a simple web application has been created, which is executable inside *Java Application Containers*, such as the one provided by *Glassfish.* An example for a mobile device has been realized using an application that can be executed on *Android* devices. Since both examples use *Java*, it is possible to use the client libraries, which come with Open Inspire.

5.2. Data Storage with NeXus

Beside the provision of a HMI for the control and visualization it is at least as important to provide storage mechanisms when it comes to scans or the recording of data histories.

Section 2.4.4 covered the topic *Short Term Storage* and *Long Term* storage in a general way. For OI, the *Short Term Storage* is in the responsibility of each individual OIM. This has the advantage that 1) the best fitting storage and processing mechanisms can be applied per OIM rather than finding an abstracted solution that has to work for all OIMs and 2) tailored interfaces for the lossless access to raw data can be provided. *Long Term Data* however need

to be stored in a consistent way to allow the data to be easily accessed using independent tools.

For diffraction measurement data, section 2.4.4 on page 49 already introduced that the *NeXus* format has been established as a standard. It is thus used for the reference implementation in this chapter.

5.2.1. API and Technology Choice

The standard way to read and write *NeXus*-files is using the *NeXus*-API. This API is part of the official *NeXus* library, which is published and regularly updated by the *NeXus*-team to reflect the latest *NeXus* development state. Although a native interface for *Java* is provided, the core library is written in *C*. This breaks the platform dependency, because the library must be compiled and bundled for each individual OS / architecture combination. For a long time, the *NeXus* library was only available in a 32 bit version. This is no problem when it is used from a C/C++ program, because even 64 bit programs are able to use the 32 bit version. For *Java* this is different. A 64 Bit JVM is unable to load a native 32 bit library so that the usage of the *NeXus* library with *Java* is strongly limited to the supported systems and the published *NeXus* builds. Beyond this it should be noted that an individual static[9] *NeXus* library has s size of more than 10 MB so that the overall size of a multi platform package quickly aggregates with each supported platform.

To circumvent these problems, *Darren Kelly* introduced the so called *NeXus Beans* [nexa, LKH06] at the NIAC[nia09] meeting 2006. The approach uses the *Java Beans* concept [Eng97] to provide a *Java* compliant way to conveniently deal with *NeXus* tree structures. The *NeXus* XML *schema* files that describe the structure are for this purpose converted to *Java* classes, which provide *getter*- and *setter*-methods to read and write stored data. The complete *NeXus* tree structure can therefore be mirrored by interlinked *Java* objects. This allows the flexible assembly of a memory based *NeXus* data model in an *Object Oriented* way. To allow the storage of this object tree to a non-volatile storage media it can be serialized to a *NeXus* compliant *XML* file.

[9] A static library integrates all dependencies in one file and does not need to reload dynamic libraries from other files. This is mandatory for native *Java* libraries because they are unable to resolve the dependencies.

The first problem with this approach was that the *XML* DTDs that were used in 2006 to describe the *NeXus* structure were unsuitable for a one-to-one *NeXus*-Beans <-> *XML* conversion. This has been corrected. The other problem is, that only the creation of XML files is directly supported. For the storage of large datasets this is an important drawback because the binary HDF format provides a faster access to data, has a seriously smaller overhead and supports compression. It is thus the better format to store large datasets.

A solution for all these problems would be a pure platform-independent *Java* implementation of the *NeXus* API that provides an object-oriented interface and allows the storage of the data in HDF-files. This is however difficult to realize because it means that the complete NeXus API needs to be ported from *C* to Java, which is a very long and complex task[10] and requires an ongoing synchronization of the *C* and *Java* development. The more serious problem is however, that there is no pure *Java* implementation with the capability to write and alter HDF5 files. The official HDF library published by the *HDF Group*[hdfb] uses a native library and the only pure implementation *NetCDF*[RD90, net] did not allow to write HDF[11].

Due to the fact that the platform independence is lost if the native HDF5 library or the native *NeXus* library is used, it is more meaningful to directly use the original *NeXus* library. This approach saves the implementation of all low-level *NeXus* functionality but requires additional work for the *object-oriented* wrapping of the *NeXus* API and the provision of a simple multi-platform solution.

5.2.2. Integration of the native NeXus Library

Since it is not possible to have a platform-independent solution, it is at least possible to provide a convenient multi-platform way using OIs native library loader (see section 4.2.2 on page 103). The *NeXus* libraries have, for this reason, been prebuilt for all major platforms and packaged in form of OILs. OIMs can now define the *NeXus* OIL as dependency and the container takes care that the correct library is automatically downloaded for the specific

[10] Especially low-level operations with elementar datatypes such as `unsigned int` and operations such as bitwise shifting are difficult to port to *Java*.Java until JDK 7 for instance does not support `unsigned` datatypes.

[11] In the meantime several projects aim to support HDF5 write support. The *Nujan* Project[muj11] for instance allows to create but not to alter HDF5 files in pure *Java*.

platform when it is requested. The administrator thus does not have to worry about which library to install, bandwidth and storage space is saved and the library does not need to be statically installed in the system path. This allows the updating of the library automatically and the ability of having more than one version installed at the same time.

5.2.3. Object Oriented NeXus Wrapper

With the OI *NeXus* library it is possible to use the full range of *NeXus* features by OIMs. Unfortunately the way how data is read and stored into the *NeXus* tree, as well as the random switching between *NeXus* branches, is rather unusual and inconvenient in *Java* due to its origin in *C*. Opening a *NeXus* file is still easy by creating a `NeXus` instance and passing the filename and a second parameter that sets the filetype (XML, *HDF4* or *HDF5*) and access mode (read / write). However, to read and write data from and to *NeXus* it is necessary to assemble structured arrays and pass them as data containers to the *NeXus* library. Further it is not possible to directly pass the information using *Java*'s standard data types and so the data generally needs to be submitted as non-typesafe *Java Objects* in combination with a parameter that sets the *NeXus* specific file type and length. And finally it is necessary to go step by step through the *NeXus* tree structure to reach a desired *NeXus*-group instead of having the ability to randomly access data by directly pointing to the path. The *NeXus Japinotes* [jap12] illustrates these specific issues in detail so that a detailed description of the *NeXus* API is omitted in favor to a short demonstration of OI's extended *NeXus* support.

OI comes with two OIMs that assist the functions of the *NeXus* OIL. They are named `org-oim-storage-nexus` and `org-openinspire-oim- storage-nexus-entities` and provide a convenient wrapper to the *NeXus* API.

The first OIM `org-oim-storage-nexus` contains general wrapper classes. These encapsulate the *NeXus* datatypes to make them type-safe and accessible in a common and object oriented way. The two main wrapper classes are `Group` and `GroupEntry`, which represent the branches and nodes in the *NeXus* tree. From the `GroupEntry` derives Scientific Dataset (SDS), which is in turn the superclass of the specialized datasets *SDSString*, *SDSNumber* and *SDSBoolean*. These all come in 4 dimensions so that an *SDSNumber3* for instance can hold three numbers

(one for each dimension). An *SDSNumber* is a class that expect a *Generic* that derives from the *Java* datatype *Number* (`class SDSNumber1D<T extends Number>`) and can so represent any possible number type in *Java*. This allows all *Java* types to be converted automatically to the according *NeXus*-datatype while achieving type-safety and hiding it from the user. To assemble a *NeXus* tree it is now possible to create a network with these classes. The root is represented by a *Group* that allows to add an arbitrary number of *Groups* or *GroupEntries* that in turn allow to add entries so that a tree can be assembled. When the tree is created, it is possible to hang it into a *NeXus* container, which serializes it to *XML*- or *HDF*-based *NeXus* files. This approach hides complexity from the user and also increases the safety due to the type-safety and a *Java*-based *Exception*-handling.

The second OIM `org-openinspire-oim-storage-nexus-entities` contains wrapper classes such as *CounterStorage*, *DetectorStorage* and *PositionerStorage* that allow to conveniently persist data of the classes that are published by the OIM *Domain Science* to *NeXus* files. Figure 5.6 demonstrates how this works in practice.

```
1  import org.openinspire.oim.storage.nexus.api.Group;
2  import org.openinspire.oim.storage.nexus.api.NexusContainer;
3  import org...oim.storage.nexus.entities.PositionerStorage;
4  import org.openinspire.oim.domain.science.Positioner;
5  ...
6
7  NexusContainer nexus = new NexusContainer("/path/to/filename.
       hd5", NexusContainer.FileType.HDF5);
8
9  Group nxEntry = new Group("entry1", "NXentry");
10 Group nxInstrument = new Group("instrument", "NXinstrument");
11
12 nxEntry.addGroupEntry(nxInstrument);
13
14 Positioner p = new ExamplePositioner();
15 PositionerStorage s = new PositionerStorage(p);
16
17 nxInstrument.addGroupEntry(s);
18
19 nexus.storeExperimentEntry(nxEntry);
```

Figure 5.6.: *Storing files to a NeXus database using the NeXus OIM*

In **line 7** an object from type `NeXusContainer` is created that provides access to the *NeXus* file */path/to/filename.hd5* and sets the format to *HDF5*. **Line 9** creates the root entry *entry1* for the experiment, which has the type `NXentry` and **line 10** a further group for the description of the instrument. This group with name *nxInstrument* is then added to the *nxEntry* group and shows how to link groups to assemble the *NeXus*-tree. Now a `Positioner` is created in **line 14** and a `Positioner Storage` in **line 15**. The constructor of the wrapper `PositionerStorage` expects a `Positioner`, no matter if it is a *Caress Positioner*, a *Simulation Positioner* or any other that implements the *Positioner* interface. This *PositionerStorage* object can now be added to a *NeXusGroup* like any other *GroupEntry* (**Line 17**). **Line 19** finally stores the root experiment entry *nxEntry* recursively to the *NeXus* file so that the complete tree is persisted to a file. All storage objects of the `org-openinspire-oim-storage-nexus-entities` OIM therefore come with a `store()` method that is called during the serialization and writes the contents of the wrapped object to the *NeXus* file.

The resulting file is a valid *NeXus* HDF or XML file that can now be opened by any HDF / XML or *NeXus* editor such as *TvNexus* (see 2.4.2 on page 43).

An important advantage of the wrapper concept is, that *Storage* wrappers can be created for every type of storage format so that it is easy to switch from a *NeXus*-based storage format to a different format just by providing new *Storage* wrappers.

5.3. Scan Implementations

With the implementation of *Devices*, UIs and *Storage* mechanisms all prerequisites are established to provide *Scans*. Through the work of the recent sections it is possible to graphically monitor the current state of all instrument components, alter them through controls and save snapshots of the current instrument state to files. To model a full-featured instrument system it is however essential to have means that allow the execution of preconfigured sequences of commands that can run over a long time, process and reduce measured data and store the results to files or databases. All of these missing requirements are covered by OI's *Scan* OIMs. Providing scan functionality as OIMs follows OI's best practices since it decouples

scan logic from specific hardware and thus allows modeling of a scan once and the subsequent reuse of it at multiple instruments that may all come with different hardware. This makes it easy to share scan algorithms with the community without limiting the *Assembly* to specific hardware. A *Demo Positioner* can therefore be replaced by a *CaressPositioner*, *McStas Positioner* or any other positioner, provided that it implements the *Positioner* interface from `org-openinspire-domain-science`.

This section gives a short survey about the general realization of scans in Open Inspire and summarizes a selected number of specialized scans.

5.3.1. External View

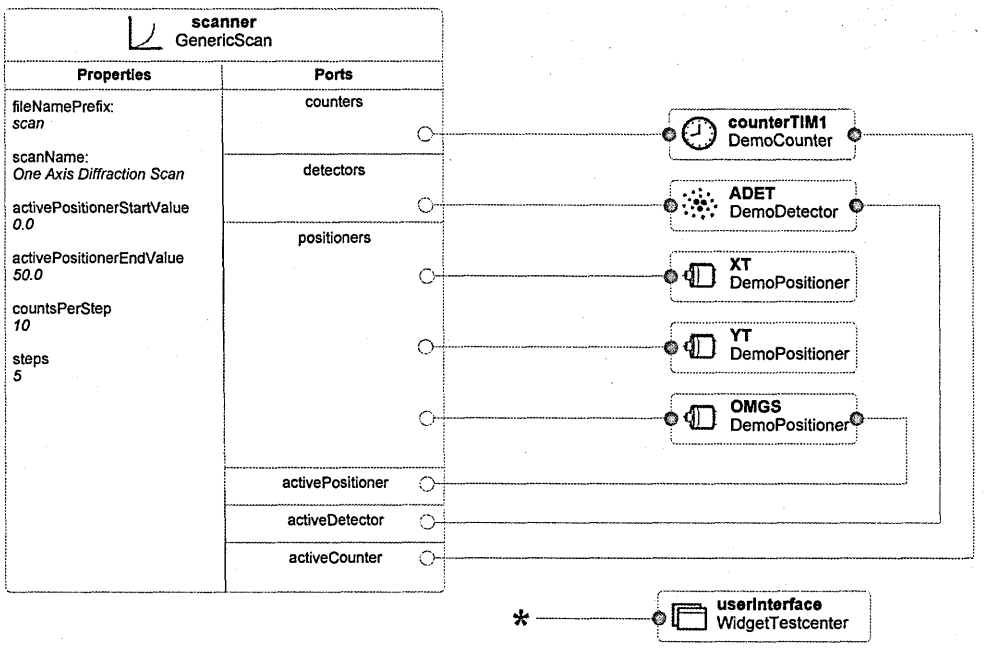


Figure 5.7.: *Genric Scan Assembly*

Figure 5.7 shows the assembly of a scan setup, which uses the OIM `org-openinspire-oim-scans-generic` to model a simple *One-Axis-Scan*[12]. This can be downloaded from the OI *Store* as well as general and a customized scan widget that is included in the `org-openinspire-oim-debug-widgets` OIM. The general widget matches all OIMs that implement the *StartStopController*, which applies for all scans. It shows a simple panel that

[12]All axes will be positioned at the beginning but only one axis will be moved during the measurement.

allows to start and stop a scan while a customized widget allows to match the advanced functions of the individual type of scan OIM to provide means to configure device relations, configuration and storage paths or scan parameters such as the number of steps and counts.

A *Scan* OIM usually comes with several *OIPorts* that allow to tether the devices that are to be controlled, queried or used to process data. The scan OIM in figure 5.7 has six *OIPorts* and six *OIProperties*. The first three ports accept multiple OIM references (see 4.3.2 on page 117) to connect an arbitrary number of *Positioner*-, *Counter*- and *Detector*-OIMs. The other three ports allow to connect an individual *Positioner*, *Counter* and *Detector* that shall be selected as active per default. The *OIProperties* are used to set scan configuration values that are either entered as preset values when a customized scan GUI is available or make it possible to even start a scan when no *GUI* or only the general *Start/Stop-Widget* is available.

5.3.2. Implementations

While different scan OIMs, when considered as a black box, differ only marginally, their internal implementation can be completely different. *Scan* OIMs with identical *OIPorts* can, for instance easily be replaced without touching the rest of the *Assembly* or transferred to other instruments. To give an impression of different scans, this section continues with a short survey to different scan engines that are made available as OIM. The overview starts with the *Generic-Scan*, which is a configurable *One-Axis-Scan* and most commonly used in neutron-diffraction experiments, continues with a *Sequencer Scan*, *Script-based-Scan* and summarizes some advanced scan approaches.

Generic One Axis Scan

Figure 5.8 shows the simplified internal realization of the *Generic Scan*[13], which can almost identically be applied to other scans. A scan generally consists of 5 steps:

INITIALIZATION: In the first step, the system performs the initialization of all connected devices and prepares the database. The example sets the initial values for all connected *Positioners*, *Counters* and *Detectors*, creates *NeXus* wrappers, which allow the storage of data to a *NeXus*-database, and sets up the *NeXus*-tree, required to hold the experiment

[13]The *Generic Scan* can be found under `org-openinspire-oim-scans-generic` inside the OI *Store*

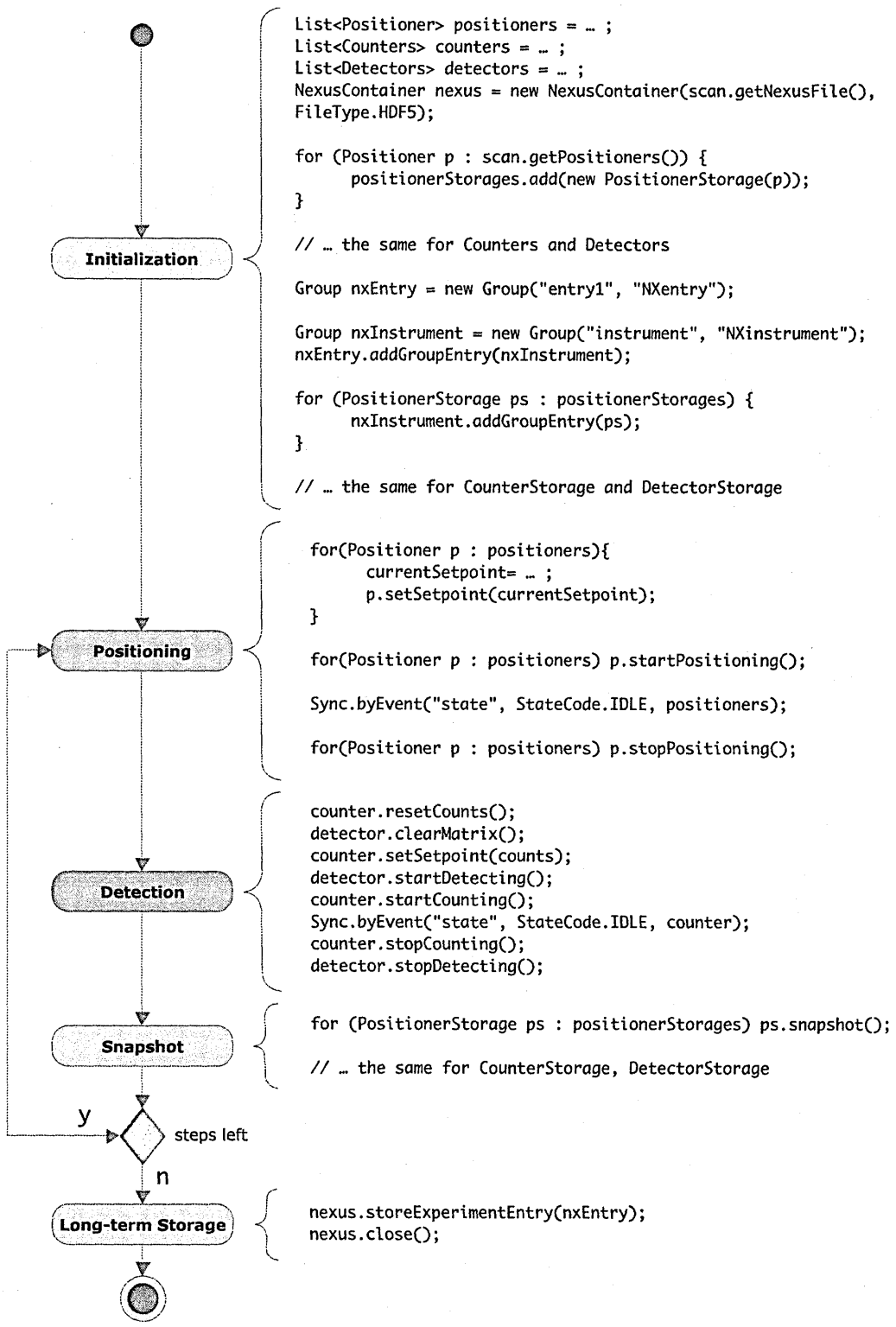


Figure 5.8.: *Simple Scan Activity*

data. If the setpoints for the individual scan steps do not depend on dynamical environment values, they can be precalculated here.

POSITIONING: The second step passes the current setpoints to the *Positioners*, triggers the positioning and synchronizes the devices. If every *Positioner* would block the program flow until it reaches its setpoint, all individual positionings need to be performed sequentially. This is common in many control systems but often consumes unnecessary time compared to a parallel positioning and furthermore prevents the performing of sophisticated scan tasks where multiple axes need to be driven at the same time[14]. To circumvent these restrictions, OI *Devices* are *non-blocking* and perform their operations in the background. This allows a parallel execution but requires some kind of synchronization, which is normally realized by polling algorithms that periodically read the current value and compare it with its setpoint and tolerances. This however causes an unnecessary high load and traffic so that some more sophisticated synchronization methods are provided by the library *oi-utils.jar*. The algorithm used in the example is called *Sync.byEvent*[15] and blocks the program flow until a number of selected devices enter a given state. The method therefor registers *Property Change Listener* for all devices, passed by the third parameter[16]. It then interrupts the program flow until all devices have changed the state given by the first parameter to the value *(StateCode.IDLE)* passed as the second parameter. This mechanisms make it is easy to create new scan modules without having to care about synchronization issues, ineffective polling mechanisms or complex comparisons, caused by the inclusion of tolerance calculations. Now it also becomes clear that a device comes with the fourth state `INIT` beside `IDLE`, `ACTIVE` and `ERROR`. Without the intermediate state `INIT`, a device that reached its target value (e.g. *Positioner* setpoint) before the synchronization started will block forever, because it is already *IDLE* when the *sync*-method is called and never sends the state change to `IDLE` that is required to continue the program flow.

DETECTION: After positioning all axes, slits and shutters to prepare the next measurement step it is time to start the detection. The example therefore resets the detector matrix and

[14] One Example is a *Continuar Texture Measurement, which measures while positioning one ore more axis.*
[15] org.openinspire.lib.util.sync.Sync.byEvent(final String eventName, final Object eventValue, final Object... monitoredObjects);
[16] The third parameter is realized as variable parameter list and accepts an arbitrary number of devices

triggers the detector as well as a counter to start counting. The already known *Sync.byEvent* method now blocks the program flow until the counter reaches a setpoint while the *Detector* counts incoming neutrons. When the *setpoint* is reached, the *Detector* and *Counter* is stopped so that the *Detector* matrix remains fixed until it is persisted to the experiment storage. It is free to the user which kind of *Counter* is used so that the detection time can be limited by a timer, the counts of a single detector or the cumulative counts of an area detector.

SNAPSHOT: Since all devices have been wrapped into a *Storage Wrapper* (see 5.2.3), the next step stores the actual values of all devices either into memory or a temporary database by calling the wrapper's *store* method. Depending on the amount of data it is either hold in memory or stored into a temporary database[17].

Until the last step of the scan is reached, the system prepares the next one, sets the current setpoints and continues with the *Positioning.*

LONG-TERM-STORAGE: When the scan loop has finished, the last step stores the temporary measurement data to a *Long-Term-Storage* such as the *NeXus* database used by the example. The system finally cleans up the temporary scan environment and the scan is completed.

Figure 5.9 shows the *Scan Widget* for the *Generic Scan* when it is executed inside the *Widget Testcenter.* The scan window on the right side allows to set a scan name and output file as well as the selection of the active *Positioner*, *Counter* and *Detector* from the list of connected devices. Devices that are connected as *activePositioner*, *activeCounter* and *activeDetector* are preselected. To configure a scan it is now possible to set the number of steps, the counts per step and the start- and end-position of the active counter. When the scan is started using the *Start*-button, the scan gives a feedback about the current state in the lower part of the window. Devices can as usual be connected to the *Widget Testcenter* and are monitored during the scan process. However, their manual control will be automatically deactivated by the scan to prevent influencing the measurement process.

[17]This could for instance be an *sqlite* database, which is at the same time lightweight and fast.[Owe06]

Figure 5.9.: *One-Axis-Scan UI in Widget Testcenter*

Sequencer Scan

While the *Generic Scan* is easy and fast to configure it is limited to one axis. Sometimes it is however necessary to run scans that drive multiple axis at the same time, deal with more than one counter or involve the sample environment for instance to start a heater or wait until a temperature is reached. These advanced functions can not be covered by the easily configurable *One-Axis-Scan*.

A solution to perform such advanced measurements is the *Sequencer Scan*, which can be found under `org-openinspire-oim-scans-sequencer`. This *Scan* OIM comes with an *OIPort* that allows the wiring of an indefinite number of different devices to it and two *OIProperties* that allow the setting of the filename of a *sequencer file* and an output file used to store the experiment results.

The *Sequencer File* is realized as an XML-file that contains a list of steps that are sequentially processed by the sequencer scan engine. Each step includes a set of commands that are applied to the connected devices. This is possible since each OIM exposes a set

of public methods through its service interface. These are mapped to the XML file so that setting the setpoint of a positioner omgs to *12.3* can be written as `<oim id="omgs" call="setpoint">12.3</oim>`. Synchronizing devices can be done with `<block-until state="IDLE">xT,yT,zT,omega</block-until>` and creating a snapshot is possible with `<nexus cmd="snapshot" parent="instrument">xT,yT,zT,omgs,detector</nexus>`.

An example sequencer file that includes these commands can be found in appendix B.2.1 on page 312. It is however likely that the syntax of the file will change to guarantee a better integration with external editors for the sequencer file and UIs that visualize the execution progress. Using the sequencer approach makes it possible to create nearly all possible variations of scans. To create nested or complex scans it is however meaningful to have an external tool that allows to create and validate the XML file. Such an editor is not part of the *PhD* thesis but could for example be directly integrated into an OI Client.

3D-Trajectory-based Scans with SScanSS

The third scan OIM can be found in the OI *Store* as `org-openinspire-oim-scans-sscanss` and allows running SScanSS files with Open Inspire. Similarly to the the *Sequencer Scan* it executes preconfigured scan sequences, but this time sequences generated with SScanSS.

SScanSS itself has been introduced in section 2.4.1 on page 39 and comes with a sophisticated graphical editor that is interesting for OI because it allows defining and calculating measurement trajectories trough specimen in a 3-dimensional way. While the tool was originally designed to create measurement sequences for the spectrometer *ENGIN-X* [OSJ+04, SDJE06, DDE+04] at *ISIS*, making the same generated scan-files run with OI opens the SScanSS-based 3D experiment planning to a wide range of new applications. Open Inspire especially benefits from SScanSS easy and quick 3D-modeling capabilities as well as the calculation, simulation and preview of complex scan trajectories through samples. SScanSS on the other hand profits from the feasibility to use the generated scan-files not only on the hardware it was developed for but also on all *Devices* that are supported by Open Inspire. This includes both real hardware such as the *Caress*-controlled instruments *E3* and

E7 as well as virtual instruments such as the *McStas*-based *Devices* that will be introduced in section 5.5.

Setting up an *OI Assembly* with the SScanSS-OIM is similar to the *Sequencer*-Scan with the difference that a SScanSS-input file is passed to the OIM in place of an XML-sequencer file. Although a *SScanSS*-setup comes with a text and an *hdf5*-file, it is sufficient to pass the text file to the OIM, because it contains a link to the corresponding *hdf5*-file. While the *hdf5*-file contains meta-information such as the facility name, instrument, SScanSS version and hardware setups, the text file delivers the parameters for the individual scan steps.

```
SScanSS project file: test.hdf
1
5
244.120       -147.726      124.596       -135.000      0.000000
236.376       -166.107      150.720       -135.000      0.000000
228.633       -184.488      176.845       -135.000      0.000000
220.889       -202.869      202.969       -135.000      0.000000
213.146       -221.250      229.094       -135.000      0.000000
X stage       Y stage       Z stage       Theta stage   mu_amps
```

Figure 5.10.: *SScanSS Scan Configuration Text File*

Figure 5.10 shows an example configuration for a scan with five steps. The SScanSS OIM first reads the last line, which contains the IDs of the *Devices* that are to be driven. If devices with the same IDs are connected to the *SScanSS* OIM, these are automatically correlated with the corresponding information in the scan file. Otherwise, it is possible to manually assign the connected *Devices* from the *OIAssembly* to the devices inside the scan file. Each line, beginning with line 4, stands for an individual scan step and each columns contains the setpoints for the corresponding device given in the last line.

Running the scan is hence not more than stepping line by line through the file, driving the devices to the setpoints and performing a detection between each positioning. The procedure of the *Positioner* and *Counter* synchronization as well as the creation of snapshots and the storage to a *NeXus*-file is thereby identical to the approach in figure 5.8. The number of counts and the counting device can be set as *OIProperty* or through the *Scan* GUI.

Custom Scan Engines

The recommended way setting up a scan is using a tool to pre-calculate all steps and to execute them with the *Sequencer-* or SScanSS- Scan-OIM. This approach allows to verify and restrict individual scan commands to prevent them compromising the system already before the scan is executed.

Sometimes it is however necessary to model scans that cannot be pre-calculated due to their complex or dynamic behavior. One reason could be that the necessity to react on changes in the sample environment shall directly influence the scan process. Other reasons are for instance scans that come with nested loops whose number of iterations are calculated just-in-time, depending on processed measurement results.

EXAMPLE 1: Figure 5.9 on page 201 shows a widget called *DSSummarizer* on the top-left side. This widget is a front-end for the OIM `org-openinspire-oim-scans-sscanss`, which can be connected to an area detector. It creates a histogram which visualizes the counts along the angles of a measured *Debye-Sherrer-Ring*. With this information it is possible to dynamically search for hot-spots and to optimize the measurement accuracy by increasing the measurement time at reflex positions and by skipping non-relevant areas.[18]

EXAMPLE 2: Another discipline that benefits from a customized scan module is the *Texture Measurement*, which has been introduced in section 2.2.3 on page 23. Originally a texture scan is performed by driving a *Positioner* in discrete steps combined with a short count period with an area detector. The problem with this approach is the quantification of sharp orientation maxima and that the frequent positioning of the sample takes up to 75% of the complete measurement time. Both problems can be solved with a scan that is able to position multiple positioners at the same time, while continuously detecting neutrons.

The first time such a a *Continuar Texture Analysis Scan* has been published was in the context of the *Diploma Thesis* by *Christian Randau* in 2007 [Ran07]. The work bases upon Open Inspire and uses an OIM to implement the *Texture Analysis Scan* as well as the *Caress*

[18] Constraints regarding the available beam-time did not allow to test this type of scan with a substantial instrument so that a detailed description is omitted. An example of a dynamically-influenced scan that does not depend on a neutron-beam can in exchange be found in section 5.6 on page 244.

Devices, introduced in section 5.4 to control the hardware of the instruments *E3*, *E7* and *StressSpec*. Different to the scan algorithms that come with *Caress*, OI is able to drive multiple *axes* simultaneously and calculate and save the results in a format that can be read by *StressTex*, a *Texture Analysis* toolbox, developed by *Ulf Garbe* FRM-II / ANSTO. The *Texture Scan* modules are in productive use at the HZB and FRM-II (Germany) and ANSTO (Australia).

EXAMPLE 3: A third example is the *Calibration Scan*, used to automatically align the components of a *Neutron Diffractometers*. This scan and its environment combines many of Open Inspire's unique concepts so that it is covered in the dedicated section 5.6 on page 244 used to give a closing summary of OI's core features.

Script-based Scans

The last type of scan that is however not published as a dedicated *Scan* OIM is a *Script-based Scan*. While the *Sequential-* and *Customized Scan Engines* already cover the full range of supposable scan problems the *Script-based Scan* nevertheless fills an important gap. The *Sequential Scan Engine* has the drawback that it does not have the same flexibility as the *Customized Scan Engine*. The *Customized Scan Engine* is however intended to be published as a ready-made OIM that typically never changes to keep the downward compatibility. This makes them both unattractive for a *rapid-scan-prototyping* approach where many fundamental changed are applied to a scan during the development or when a scan shall be modeled in short time without the aim to ever publish it.

For these scan setups it is possible to make use of *Java*'s JSR 223 : Scription for the Java Platform, which allows the running of a large number of scripting languages directly in *Java*. Due to *Java Scripting Host* it is already possible to write an OIM in any of the supported scripting languages, because they compile to *Java* class files that can be packaged to an OIM in the same way as any other OIM. Currently OI however does not provide the toolchain and IDE integration so that the build and packaging process needs to be done manually. This

future aim will provide the ability to write complete modules in languages such as *Jython*[19], *Groovy*, *Java Script* or *Ruby*.

The advantage of the approach is, that a user can quickly write a scan sequence by using the full feature set of the selected scripting language. The user thereby has access to all devices that are connected to the OIM from any of the scripting languages. To make the usage still easier it is moreover possible to auto create a script template that already binds the connected devices to local variables. A user can now easily alter the script file while *Java*'s *just-in-time-compiler* always updates the generated *Java* class files to reflect the latest script changes when the OIM is restarted.

The scripting approach however also has several drawbacks. The first is the problem that it is nearly impossible to prevent a script from gaining access to the system by bypassing security mechanisms. Opening the system for arbitrary scripts consequently breaks the safety of the system and should be avoided. Another problem is the violation of OI's main principles such as the aim to provide well-tested encapsulated functionality, which can be shared with the community. *Scripting* OIMs will not be officially part of OI until a well-tested toolchain and the ability to execute OIMs inside individual sandboxes is provided.

5.4. Hardware Control with Caress

Section 5.1 covered how to visualize the experiment progress using a GUI, section 5.2 covered how to store measurement data to the unified NeXus format and 5.3 introduced the way how scans can be realized with Open Inspire. All previous examples used the simulated demonstration devices *DemoCounter*, *DemoDetector* and *DemoPositioner* instead of real hardware. This section now fills this gap and shows how to replace the *Demo Devices* in figure 5.7 on page 196 with OIMs that act as a proxy for real hardware.

It is a strong demand for Open Inspire to be able to use existing legacy hardware with the highest possible flexibility and least possible effort. This section shows how to make use of OI's abstraction mechanisms to allow the changing of the type of devices without touching

[19]Especially *Jython* as the *Java* equivalent to *Python* is interesting since *Python* has been declared as the standard scripting language for neutron experiments at the NOBUGS 2008 [nob]

the rest of the *Assembly* and also how to build a bridge to existing hardware systems that were intentionally designed not to be controlled by third party systems.

For this example the control system *Caress* has been chosen. It is a good starting point since it already supports more than 100 different hardware components, which can be directly used with Open Inspire after the implementation. With the support of *Caress*-based hardware it is possible to model fully functional diffraction scans, and use the scans introduced in section 5.3 on real instruments.

The realization of *Caress* support for OI is a central requirement [☒ 4.2.3] and one of the fundamental parts of the thesis so that this section covers the topic in a slightly more detailed manner. *Caress* follows different design principles than OI and hence gives a good insight into the procedural method required to get differently designed systems to work hand in hand.

5.4.1. A survey of Caress

Caress[RS07][Sau07] is an Instrument Software for Neutron Scattering Instruments, used to control nearly all instruments at the Berlin Neutron Scattering Center (BENSC) (HZB) and some instruments at FRM-II (TU Munich). The development of *Caress* started in 1993 and the software has been continually extended to a codebase of about 164,000 lines and more than 130 device drivers. These include bus drivers for *GPIB/IEC*, *Camac*, *VME* and *TCP/IP*, drivers for single, linear and area detectors, several commercial and in-house developments of motor controllers as well as sample environment drivers. Recently drivers were added that allow the accessing of *Labview*, *TACO* and *TANGO* devices.

The Distributed Caress Environment

The network topology used by *Caress* is organized in the form of a client server system with one client and one or more servers. The central part of the system is the client which runs on a *UNIX- / Linux*-based workstation and is itself simply known under the general name *Caress*. The servers are arranged around the *Caress Client* and each one publishes a defined interface that allows access to a specific subset of instrument hardware. The communication solution used is the *object oriented* Common Object Request Broker Architecture (CORBA)[OMG04]. Each instrument thereby owns its own Object Request Broker (ORB) which acts as a name

service and registry for distributed objects. Figure 5.11 shows the setup of the neutron

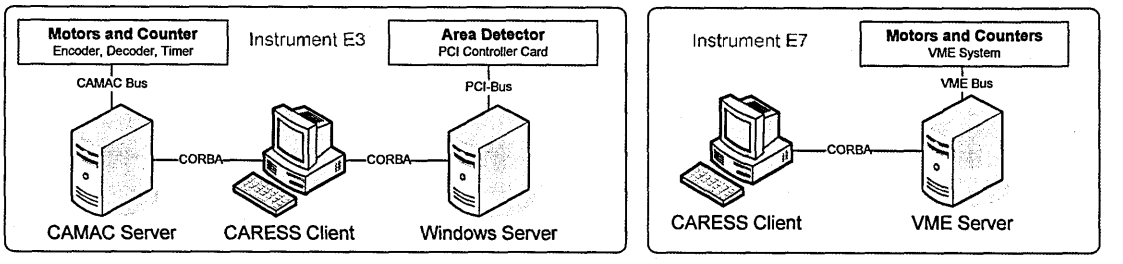


Figure 5.11.: *Caress Network at Instrument E3 and E7*

diffractometers *E3* and *E7* at BENSC and *StressSpec* at FRM-II in Munich. Although all instruments are nearly identical in their geometry, number and kind of movable axes, counters, detectors and scan functionality, they vary in the way they access the hardware. Instrument *E7* and *StressSpec* use one *VME* based server both for the hardware access and the operation of the ORB while the devices at instrument *E3* are split between two servers and the ORB runs on the client machine. The first server handles the detector, which is connected using a *PCI* card and the second server uses a *CAMAC* system to manage the remaining devices. These differences are important when it comes to the issue of allowing each instrument to be addressed independently.

The Caress Client

The *Caress Client* is the central element of *Caress* controlled instruments. It is responsible for the initialization and control of all devices, hosts the device configurations, provides a GUI and CLI, performs scans and stores measurement data.

Configuration Files

Whenever experiments are started using *Caress*, configuration files are processed, which contain initialization parameters for the experiment and instrument hardware. Two of these files for instrument *E7* are listed in Appendix B.3. The first file *hardware_modules_e7.dat* on page 314 is used to initialize devices with specific parameters, define the hardware servers and conditionally enable devices for a selected scan setup. The second file *params_e7.com* on page 315 is used to setup parameters such as software limits, tolerances or offsets for

already initialized devices. Both files act as resource for Open Inspire's module configuration parameters.

The User Interface

Caress offers the choice between a Command Line Interface (CLI) and a Graphical User Interface (GUI) that both basically provide a prompt for Caress commands. The range of these commands begins with setting experiment properties, device setpoints or preset values, continues with the positioning of axes and ends up with performing simple one axis scans to complex multiple axes scans. A user typically uses the GUI that extends the CLI with more user friendly dialogs for scan setups and device configurations. Figure 5.12 shows the three windows that are opened when the GUI is started. The first window shows *Monicar*, a monitoring tool that shows constantly updated information about device values such as counts or motor positions. The second window is the main window and provides a prompt, a command history and menu access to configuration dialogs. The last window is a *PV-Wave*-based viewer for detector data.

Caress Documentation

Developing a bridge between Caress and Open Inspire requires gaining an insight into the Caress internals, which is in some situations difficult due to the lack of information. *Caress* is developed and maintained by a small group of people and not by a public developer community that shares information by forums, mailing lists or published information. The *Caress* team itself alluded to this issue in their *Caress* presentation summary from the SEI in 2007 [RS07] (page 22): *"The biggest deficit is the incomplete documentation"*. One important document provided by the Caress developers is the device parameterization manual [DPSR09], which can be found inside the *doc*-folder of every *Caress* installation. This file contains all valid device initialization parameters and is similarly used by Open Inspire to setup the *Caress* OIMs. Further files regarding the usage of the CLI or usage scenarios can be found inside the *doc*-folder of *Caress*'s source code distribution.

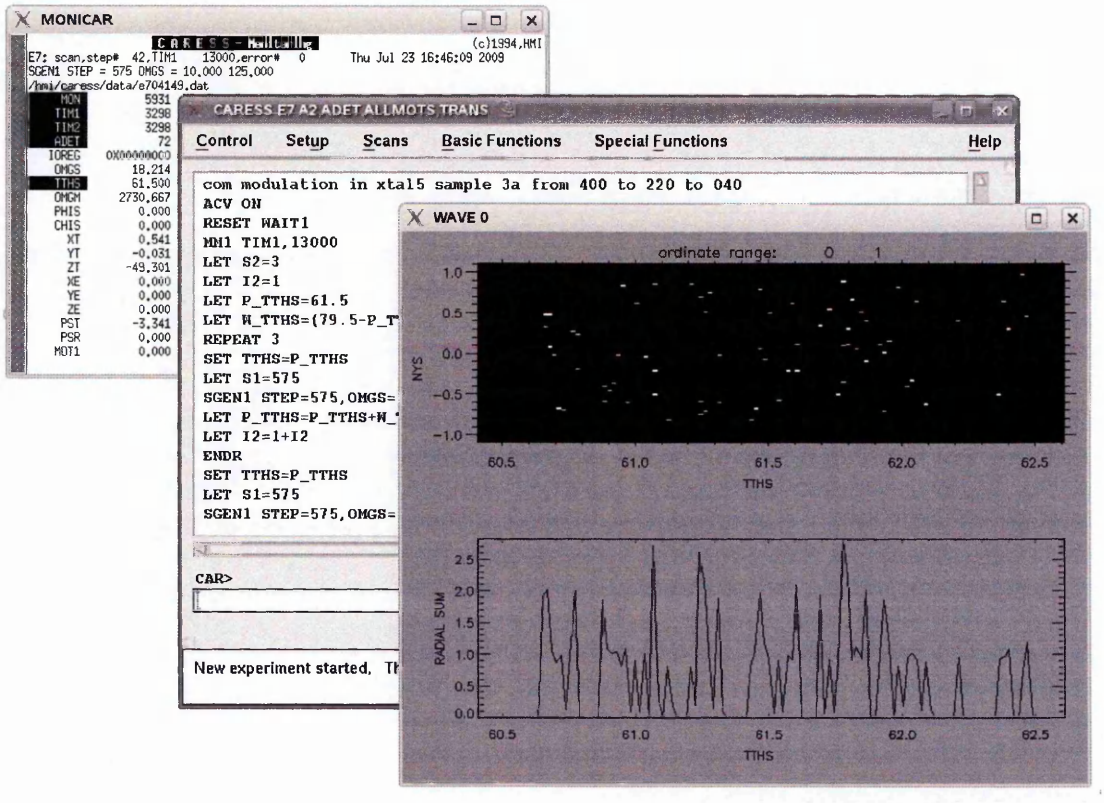


Figure 5.12.: *The Graphical User Interface of Caress*

During the development of the *Caress* OIMs it emerged that a combination of Caress documentation combined with information gained by reverse engineering of Caress sources[20] and discussions with *Caress* Team Members is a way to cope with the majority of problems. The most time consuming task is the reverse-engineering of the *Caress* sources that are responsible for the hardware access.[21] A lack of comments also hinders the decoding of the dynamical behavior of stateful operations.

5.4.2. General Design Decisions

Implementing the *Caress* support requires some preliminary steps. This includes the setup of a test environment, the communication between Open Inspire and *Caress* as well as the base configuration of the *Caress* OIMs for Open Inspire.

[20] The Caress sources are stored in a HZB wide subversion repository
[21] Hardware access related sources are stored in the *capi* folder of the source code distribution

The Test Environment

Preliminary to the development it was mandatory to find a qualified test environment. By reason that all instruments are in regular service it is essential to have an environment that reproduces the real instrument as accurate as possible. A simulation of the whole environment is not beneficial due to the high complexity and time effort so that a real *Caress* instance is a more meaningful choice in this situation. During the development of the OIMs, the test environment has been improved several times to reduce the dependency on the actual instruments and to allow critical tests that could otherwise harm the hardware. *Caress* at first only ran on *DEC Alpha Stations*, which required the use of real experiment hardware. Early tests have been performed using the rarely used but unfortunately incomplete instrument *E2a* (HZB). After the first stable Open Inspire version was released, these tests were carried out at the instruments *E7* and *StressSpec* during reactor rest periods. Later on, the *Caress* team released a version that also runs on *Linux* systems. This version is installable in a virtual machine and can simulate the instrument without real hardware. The new version, for the first time, allows the testing of *Caress* setups on standard PCs so that critical functionality can be executed in a simulated environment. This prevents harming the instrument due to malfunctions. With this step it also became possible to run experiments from office networks that where not possible before, due to the random CORBA ports that could not act as tunnels between the different VLANs at the HZB. A further step in early 2009 finally made it possible to port the simulation server, with some help of the *Caress* team, to *Mac OSX*. This port is used for all dynamical tests such as the realtime behavior of method calls or network throuput including latency times. The network monitoring tools used are *netcat* and *wireshark*, which run with an extension for the monitoring of *Caress' CORBA packets.*

Selection of a communication solution

A mandatory precondition for the usage of legacy systems is the identification of a suitable communication approach between Open Inspire and the legacy system.

The first considered way is the reuse of a *Caress* interface, developed for *Frack* within the scope of a diploma thesis in 2005 [Fle05]. This external interface has become, in the meantime,

an integral part of the *Caress* API [22] and allows the sending of commands to *Caress* and the querying of data such as a detector matrix or motor positions. It is practical for simple control scenarios but has some essential drawbacks that prevented its usage with Open Inspire: 1) The interface does not support the initialization of devices and so prohibits operating *OI* without a parallel *Caress* installation, 2) it is limited to commands that are interpretable by *Caress*' internal runtime engine and prevents integrating functionality beyond the scope of *Caress*. These commands furthermore send asynchronously, without associated return values, which prohibits allocating commands to return values, 3) finally, only a subset of all data can be read and this with low performance, which is not sufficient to tap the full potential of the Caress hardware.

A second approach is the extension of *Caress* by a new network service. This service could be tailored to the needs of Open Inspire and provide all required information and access to the hardware. A relevant advantage is the reusability of *Caress* code and the option to directly optimize the interface and handle internal state machine and invisible dependencies on the server side. Both, the extension of the external interface and the development of a new service require changes to *Caress*, which results in a high maintenance effort and compatibility problems. It is the reason why the OI guidelines state that legacy software should be usable with OI without altering the system itself. [☒ 5.3.2] [☒ 5.3.6]

A third approach is the direct usage of the CORBA based services provided by the *Caress* hardware servers. This assures the highest flexibility without changing any parts of *Caress*. The whole functionality provided by the hardware remains preserved and Open Inspire can be used as standalone replacement for *Caress*. The cost of this approach is that some functionality of the *Caress Client* needs to be reproduced by Open Inspire. This includes the initialization of hardware as well as the providing of a timing source and adapters for the conversion of *Caress* data and operations to the OI infrastructure.

All further work is based on the third approach due to its flexibility and conformance to the stated equirements of Open Inspire.

[22] The external interface can be found in the */src/carapi/carapi.java* folder of the *Caress* sources

The Caress OIMs

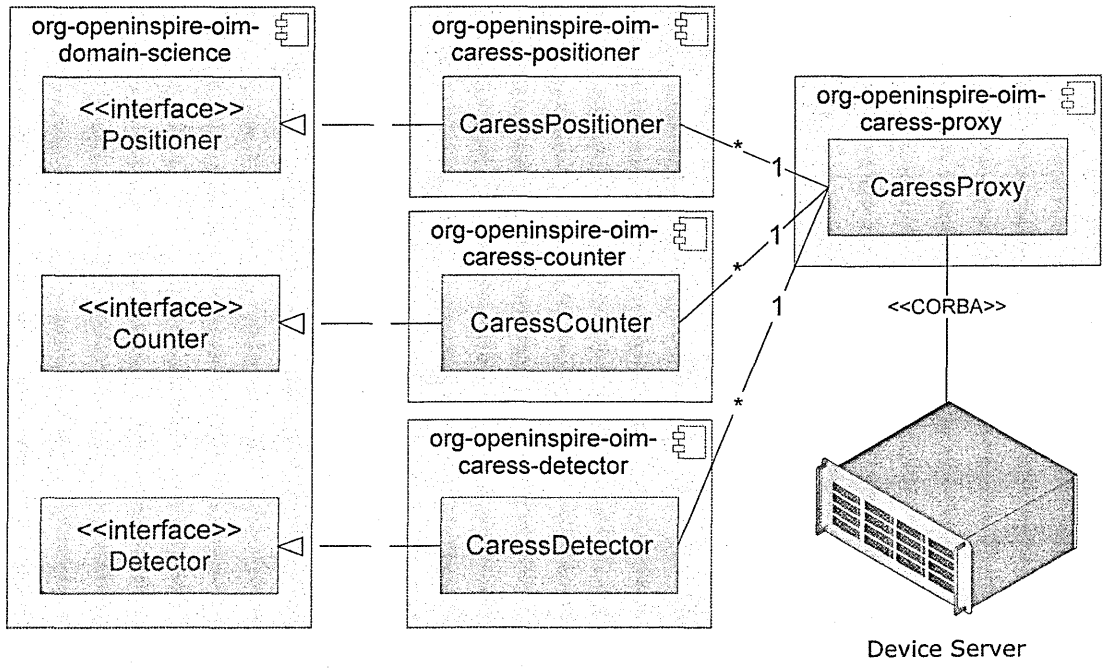


Figure 5.13.: *Implementation hierarchy of the Caress OIMs*

Modeling diffraction scans requires access to *Positioners*, *Counters* and *Detectors*. The prescribed way to realize this functionality using OIMs is the implementation of the appropriate interfaces from *org-openinspire-oim-domain-science*, illustrated in figure 4.20 on page 126. This assures the compatibility between all already existing and future subsystems that use the same interfaces. Figure 5.13 shows the OIMs *org-openinspire-oim-Caress-positioner*, *org-openinspire-oim-caress-counter*, *org-openinspire-oim-caress-detector* and the additional OIM *org-openinspire-oim-caress-proxy*, which is covered in the next section.

5.4.3. The Caress Proxy OIM

The *Caress Proxy* is a complementary construct that has been introduced as a solution for various design issues. The obvious approach, where each OIM represents exactly one device (such as a *Positioner*, *Counter* or *Detector*) is not directly realizable due to hidden internal dependencies between *Caress* Devices. One example is the *Counter* that needs to know all other counters to set them to slave mode before counting in master mode. An other example

is the *Detector*, whose hardware is connected with the primary counter and starts and stops counting with it. Introducing links between all devices is not an option because it increases the complexity and abates the maintainability of the *Assembly*. Using dependencies between devices by static classes or singleton objects on the other hand creates hidden dependencies and causes unmanageable side effects.

Here the *Caress Proxy* OIM not only solves this problem but also presents several design advantages. As already stated in section 5.4.1, some setups require links to more than one hardware server. Using multiple *Caress Proxies* allows connecting to an arbitrary number of hardware servers and address multiple instruments simultaneously from within the same *Assembly*. Furthermore, configuration redundancy in the *Assembly* can be reduced with the benefit of a better maintainability, because only the *Proxy* needs to be configured with the connection parameters and not the devices themselves.

Configuration of the Caress Proxy

The Caress Proxy provides all required functionality to establish a connection with a CORBA based Caress service from within Open Inspire.

```
1 <oim id="CaressProxy" type="org-openinspire-oim-Caress-proxy"
    version="1.3.0">
2   <property name="ORBHost" value="localhost" />
3   <property name="ORBPort" value="2809" />
4   <property name="objectPath" value="example.context/absdev.object"/>
5   <property name="slowDeviceTriggerInterval" value="10"/>
6   <property name="normalDeviceTriggerInterval" value="1"/>
7 </oim>
```

Figure 5.14.: *Initialization of a Caress Proxy OIM inside an Inspire Context File*

Figure 5.14 shows the configuration for a *Caress Proxy* within an *OI Assembly*. The first *OIProperty* `ORBHost` specifies the name of the server, which hosts the *CORBA Naming Service* and the optional property `ORBPort` the port[23] of the registry. The third property `objectPath` specifies the path to the CORBA object in the registry that provides all functionality that is required to communicate with the hardware server. The two remaining optional proper-

[23]The default port of the CORBA registry is *2809*.

ties `slowDeviceTriggerInterval` and `normalDeviceTriggerInterval` are finally used to configure the timing source that triggers the *Caress* hardware server.[24]

Connecting the CORBA Naming Service

Figure 5.13 on page 213 shows, that the *Caress Proxy Module* communicates with a CORBA service, which is registered by the *Caress Device Server.* Connecting the service is done directly from within the *Java* code, since *Java* is already equipped with built-in CORBA functionality. Using *Java*'s CORBA API offers a platform independent solution without the need for external libraries but some necessary configuration work due to different implementations of the comprehensive CORBA specification. More details can be found in the source code under *org.openinspire.oim.Caress.proxy.api.DeviceServer.*

Communication with the Caress Hardware Service

After establishing a connection to the Object Request Broker (ORB), the next step is calling the remote methods on the hardware server. To achieve this, Open Inspire resolves the *Remote Object found* at the location, passed by the `objectPath` configuration property (see figure 5.14) and narrows it regarding the interface definition of the *Caress Hardware Service.* While resolving and narrowing is realized in a few lines of source code in the constructor of *CaressAbstractDevice* inside the *API*-Package of the *CaressProxy*, the mechanisms used to create and handle the remote interfaces requires a more detailed explanation.

Beside the ORB that facilitates the communication between objects, CORBA provides a so called Interface Definition Language (IDL) that is used to describe interfaces between CORBA instances in a platform and language independent way. These interfaces are not used to implement a service but as a kind of blueprint that is required to generate so called *client stubs* or *server skeletons* for nearly all existing programming languages. A native *Caress* environment implements the *server skeletons* for hardware servers as well as the *client stubs* for the *Caress* client, both in *C/C++*. |aclOI requires *client stubs* for *Java*, which are described in detail after examining the *Caress Interface Definition.*

[24]The default of 1 and 10 seconds normally needs no modification but can be used to fix timing issues.

The Absdev Interface Definition

The interface definition, which is used by the *Caress* hardware server can be found in Appendix B.3.2 on page 315. The file is part of the Caress source code distribution and stored as *absdev.idl* in the *omnisrc* folder. The figures 5.15 on page 216, 5.16 on page 5.16 and 5.17 on page 5.17 show tables with all operations provided by the *Absdev*-Interface in a clearly arranged form. Cells with dark background mark global operations while light cells illustrate device based operations.

Initialization, Shutdown and Configuration Operations

	A	B
1	long **init_system_orb**(in long kind_of_init_system, out long system_status);	long **release_system_orb**(in long kind_of_release_system);
2	long **init_module_orb**(in long kind_of_init_module, in long module_id, in string module_line, out long module_status);	long **load_module_orb**(in long kind_of_load_module, in module_info_seq_t module_info_seq, in long event_number, out long event_used, out long module_status);
3	long **char_loadblock_module_orb**(in long kind_of_loadblock_module, in long module_id, in long start_item, in long end_item, in long item_type, out long module_status, in char_data_seq_t data_seq);	long **short_loadblock_module_orb**(in long kind_of_loadblock_module, in long module_id, in long start_item, in long end_item, in long item_type, out long module_status, in short_data_seq_t data_seq);
4	long **int_loadblock_module_orb**(in long kind_of_loadblock_module, in long module_id, in long start_item, in long end_item, in long item_type, out long module_status, in int_data_seq_t data_seq);	long **float_loadblock_module_orb**(in long kind_of_loadblock_module, in long module_id, in long start_item, in long end_item, in long item_type, out long module_status, in float_data_seq_t data_seq);

Figure 5.15.: *Caress Absdev Init and Shutdown Operations*

The first figure 5.15 on page 216 shows all operations used to initialize, shutdown and configure the device server and related devices. `init_system_orb` and `release_system_orb` need to be called globally to initialize and release the whole device server. The operations `init_module_orb` are called to register and initialize devices that are connected to the device server and are afterwards ready to receive further commands. Beside a unique ID, a specific initialization String needs to be passed that requires to follow the rules published in the *Hardware Module Documentation* [DPSR09]. Examples of the initialization parameter can be found in figure 5.23 on page 227, 5.25 on page 230 and 5.27 on page 233 by consulting the property `deviceParameter`. The remaining methods except of the `load_module_orb`-operation are not required for the implementation of *Positioner*, *Counter* and *Detector* functionality. The operation `load_module_orb` is certainly used by *Positioners* and *Counters* to pass setpoints to positioners and counters.

Start, Stop and Drive Operations

The second figure 5.16 on page 218 illustrates operations used to start, stop and drive modules. `start_acquisition_orb` is a global command that is called before starting new measurements and with a different `kind_of_start_acquisition` parameter prior to new measurement steps. `stop_acquisition_orb` causes the opposite of `start_acquisition_orb` and pauses or stops measurements. It is important to keep in mind that *Caress* uses internal state machines and makes assumptions about involved devices, which are altered by the measurement without explicit configuration of these dependencies. The starting of an acquisition automatically starts all registered counters and reads back detector values until a selected master counter reaches its target value. The third global operation `stop_all_orb` immediately stops all registered devices and is used whenever a complete and fast shutdown is required. To really stop all modules, `stop_all_orb` needs to be called multiple times with different parameters.

The device specific counterpart of `stop_all_orb` is the operation `stop_module_orb` which requires the unique id of a module and only stops this selected module. It needs to be called when a positioner reaches its end position or when a counter finished counting to its target value. The last operation `drive_module_orb` is intended to start driving some drivable module. This method is, in the majority of cases, used to start moving a positioner to a target value that is passed by a parameter, which has been encoded in a `module_info_seq`-type.

	A	B
5	long **start_acquisition_orb**(in long kind_of_start_acquisition, in long run_no, in long mesr_count, out long acquisition_status);	long **stop_acquisition_orb**(in long kind_of_stop_acquisition, out long acquisition_status);
6	long **stop_all_orb**(in long kind_of_stop_all, out long stop_all_status);	long **stop_module_orb**(in long kind_of_stop_module, in long module_id, out long module_status);
7	long **drive_module_orb**(in long kind_of_drive_module, in module_info_seq_t module_info_seq, in long event_number, inout long calculated_timeout, out long event_used, out long delay_status, out long module_status);	

Figure 5.16.: *Absdev Start, Stop and Drive Operations*

Read Operations

The last table of operations, labeled Figure 5.17, can be found on page 219 and represents methods, used to read back data from devices. `read_module_orb` and its global counterpart `read_allmodules_orb` are used to read arrays with values from one device or accordingly a composed array of values from all changed devices.

	A	B
8	long **read_allmodules_orb**(in long kind_of_read_allmodules, inout module_info_seq_t module_info_seq);	long **read_module_orb**(in long kind_of_read_module, in long module_id, inout module_info_seq_t module_info_seq);
9	long **readblock_params_orb**(in long kind_of_readblock_module, in long module_id, inout long start_channel, inout long end_channel, inout long channel_type);	
10	long **char_readblock_module_orb**(in long kind_of_readblock_module, in long module_id, in long start_channel, in long end_channel, out long module_status, inout char_data_seq_t data_seq);	long **short_readblock_module_orb**(in long kind_of_readblock_module, in long module_id, in long start_channel, in long end_channel, out long module_status, inout short_data_seq_t data_seq);
11	long **int_readblock_module_orb**(in long kind_of_readblock_module, in long module_id, in long start_channel, in long end_channel, out long module_status, inout int_data_seq_t data_seq);	long **float_readblock_module_orb**(in long kind_of_readblock_module, in long module_id, in long start_channel, in long end_channel, out long module_status, inout float_data_seq_t data_seq);

Figure 5.17.: *Absdev Read Operations*

The parameter `kind_of_read_modules` or `kind_of_read_allmodules` specifies which kind of devices should be returned. *Caress* distinguishes between slow, normal and fast devices which require different wait periods between consecutive reads. The remaining methods allow the reading of parameters and values of a block module such as a detector histogram. Here

the caller needs to know the datatype of the block that should be read and with it selects the corresponding `readblock`-operation. Reading the wrong datatype or a wrong blocksize results in a crash of *Caress*.

Wrapping the Caress Communication Architecture

The disquisition regarding the *absdev* shows that only one communication interface is used to handle all conceivable forms of communication between a caress client and hardware servers. The outcome is that the *CaressProxy* also requires only one *client stub* per *CaressProxy* that has been generated according to the *absdev*-Interface Definition. This stub is generated using the tool idlj (Interface Definition Language to Java Compiler), which is part of each Java Development Kit (JDK) and used to create all interfaces and classes required to communicate with a CORBA object from *Java* code.

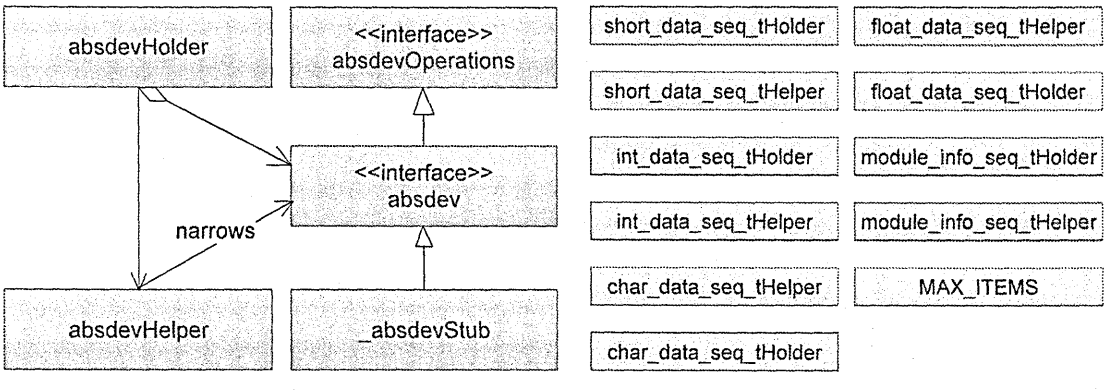


Figure 5.18.: *The generated client stub files by idlj*

The created code has been refactored to match the OIM guidelines and can found as part of the *Caress Proxy* OIM inside the package *org.openinspire.oim.caress.proxy.absdev* and the package *org.openinspire.oim.caress.proxy.absdev.types*. A class-diagram with all generated classifiers is shown in figure 5.18. The most important part of the generated code is the *absdev*-interface, which provides all required methods to access the CORBA remote object. All remaining classes and interfaces either define operations and structures (*absdevOperations* and the entities on the right hand side) or store, cast or serialize and deserialize objects (*absdevHolder*, *_absdevStub* or *absdevHelper*).

CORBA is a middleware that allows the distribution of complete object hierarchies but *Caress* does not make use of this functionality and exclusively uses the *absdev*-object to publish remote functionality. This decision can be justified by the evolution of *Caress* and the limited scope for making changes without causing compatibility issues.

Open Inspire generally allows the use of this *absdev*-reference in each *Caress OIM*, but there are many reasons in favor for introducing an object oriented layer around the *absdev*-object. Implementing the wrapper functionality at a central place inside the proxy OIM allows all other *Caress* OIMs to use the wrapper instead of the complex *absdev*-interface and bundles the complexity into one module, which can be maintained by a *Caress* specialist. Beyond that it allows consistent exception handling, logging, and information hiding that disallows calling functions that are not implemented for a specific module and would otherwise crash *Caress*. Beside these stability improvements the redundancy of code is reduced due to a centralization of common functionality. Sharing this functionality also improves the maintainability and time effort, required to implement support for new devices. A further issue is that *Caress* uses internal state machines that can lead to different behaviors of the same operation depending on a current internal state and it is not possible to ensure that a call to an operation at one device does not influence other devices. To handle these dynamical issues, a superordinate manager needs to assure the dynamical consistency of the object system. All of these functionalities are provided by the *CaressProxy* and are illustrated in the following sections.

Initialization and Device Registry

Before a new version of *Caress* was released in 2009, *Caress* used an allocation table with unique entries for device IDs and corresponding devices. With this definition it was clearly defined that a device with id *XT*, *YT*, *ZT* or *OMGS* is a positioner, *TIM1*, *TIM2* and *MON* is a counter and *ADET* is a detector. A copy of this allocation table allowed Open Inspire to address known devices, that were preinitialized by *Caress* and even to use Open Inspire side by side with *Caress* in a passive mode. In 2009 these unique IDs have been retired and replaced by dynamic IDs. This decision was momentous for Open Inspire, because there is no way left to externally figure out the dynamic ID for a device. Even a new feature that

allows a list of all registered IDs to be obtained does not solve the problem as there is no means to obtain the kind of device from the interface.

To reflect the changes in the new *Caress* version, the *Caress Proxy* has been refactored to support the new *Caress* Design and its dynamic IDs. Prior to version *2009-m3*, the old *Caress* API can be used while all later versions are now compatible with the latest *Caress* builds. The following sections cover OI's new *Caress* support beginning with version *2009-m5*.

A simplified survey of the *Caress Proxy* architecture in figure 5.19 shows the general structure of the *CaressProxy*-OIM.

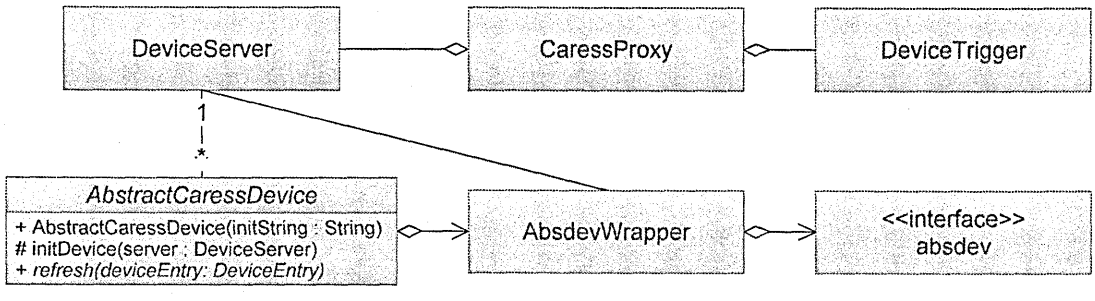


Figure 5.19.: *Survey of the Caress Proxy OIM*

Whenever a new *CaressProxy* is initialized, it creates a *DeviceServer*- and *DeviceTrigger*-instance that are configurable by properties passed by the *Assembly* parameters (see figure 5.14 on page 214). While the *DeviceTrigger* exclusively provides a timing source, which is covered in section 5.4.3, the *DeviceServer* provides more functionality. It is responsible for the connection to the ORB, initializes and releases the hardware server and acts as device registry. Once a connection to the ORB has been successfully established, it creates an *AbsdevWrapper*-object, which gains an instance of the *absdev*-object from the ORB and stores it locally.

This *AbsdevWrapper* is the new gate to the hardware server. It provides the same functionality such as the *absdev*-object, but wraps it in user friendly calls, introduces *Exception-Handling*, *Information Hiding* and follows *UML/Java* specific coding guides such as names in *Camel Case Notation.* A detailed description of all wrapper operations goes beyond the scope of

this survey. More information can be found in the *AbsdevWrapper*-class inside the package *org.openinspire.oim.caress.proxy.api* or as part of section 5.4.3.

The last important classifier in diagram 5.19 is the *AbstractCaressDevice*. Whenever a module such as the *Caress Positioner*, *Caress Counter* or *Caress Detector* needs to access the remote hardware, it creates a class that inherits from *AbstractCaressDevice* and registers this object at the *DeviceServer*. The *DeviceServer* creates a unique ID for the device, initializes the device by passing its initialization parameters to the *hardware server* and registers the object inside a local device registry. This procedure simplifies the development of modules because nothing is required except the initialization *String* and an *Object* that inherits from *AbstractCaressDevice*. A concrete device can now extend the functionality of the *AbstractCaressDevice* while having a valid instance of the *AbsdevWrapper*. Concrete examples related to the extension of the *AbstractCaressDevice* can be found in section 5.20.

Event Handling

Open Inspire uses object-oriented event mechanisms that are based on *Property Change Events*, which are relayed between OIMs. The *absdev*-design however does not provide interrupts nor callback mechanisms that are directly bridgeable to OI's event mechanism. The problem is hence only solveable by a periodic polling of the *Caress* devices and the event-based propagation of changed values to OI's event listeners.

The *absdev*-interface provides the operations `read_module_orb` and `read_allmodules_orb`, which allow to read current values and states of devices directly from the device server. The fact that every device that extends *CaressAbstractDevice* can also access the wrapped `read_module_orb`-operation could lead to a design where each *OIM* periodically reads the module data and delegate changes to listeners. This approach would generally work but evokes several problems. *Caress* sets specifications concerning the time between subsequent calls of the `read_module_orb`-method. Asynchronous calls from multiple OIMs however can not take care about these timing demands.

The solution for these problems is a global mechanism that periodically calls `read_allmodules_orb` instead of `read_module_orb`, decodes the response and redirects the data to the appropriate device. Calling `read_allmodules_orb` requires two parameters. One parameter is the reference to a `module_info_seq_tHolder` that has been generated by idlj and one parameter is an integer that specifies the kind of devices that will be covered later on.

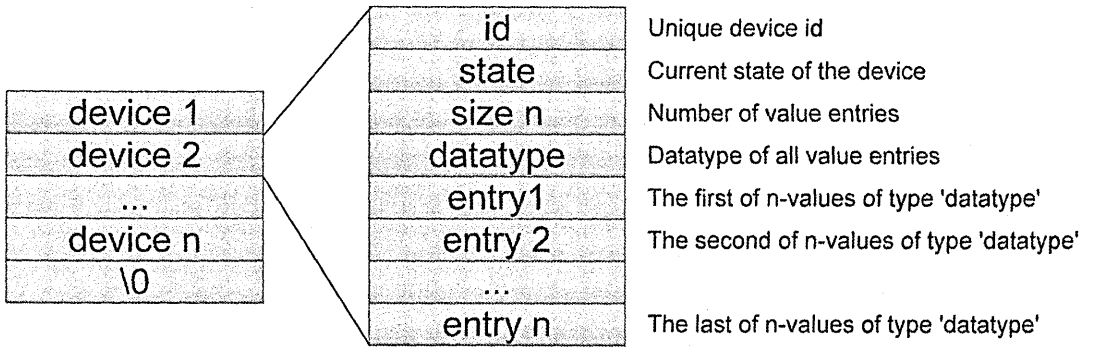


Figure 5.20.: *The Module Info Sequence*

Figure 5.20 shows the content of a `module_info_seq_tHolder`-object after calling the method `read_allmodules_orb`. It contains an array of CORBA *Any*-Objects that can contain any kind of data and need to be casted to useful types such as *int*, *char* or *double*. While `read_module_orb` returns one array with entries, illustrated in figure 5.20 (right hand side), calling `read_allmodules_orb` wraps up the entries of all devices in a superordinate array (left hand side), which is terminated by a binary null.

Delegating these Any-Arrays directly to the registered devices requires that every listener has to interpret, decode and cast the data first, which is error prone, non-intuitive and redundant. To resolve these issues, `read_allmodules_orb` has been wrapped by the method `getDeviceEntries` inside the *AbsdevWrapper*. This method returns a *DeviceEntries*-Object, which contains a *DeviceEntry*-list.

The next figure 5.21 shows the class *EntryFactory*, which is responsible for the creation of *DeviceEntry*-Objects. It accepts *Any*-Arrays of variable size from `read_allmodules_orb` and returns user friendly *DeviceEntry*-objects. It also shows, that *DeviceEntry* is an abstract class and needs to be extended by a concrete implementation such as a *DoubleEntry*, *IntegerEntry*

or *StringEntry*. Here the polymorphic capabilities of *Java* come into operation that allow inserting objects that inherit from *DeviceEntry* into generic collections of the type *DeviceEntry*. Before this is done, the *EntryFactory* interprets the passed *Any*-Array, reads the *datatype* entry of each device and creates the corresponding *DeviceEntry* before inserting the casted values. A *DoubleEntry* can so act as a container for *REAL*-values, a *IntegerEntry* for *INT*- and *LONG*-values and a *StringEntry* for *CHAR*- and *STRING*-values. This solution ensures that no OIM needs to take care about complex data structures but can simply read the relevant data by intuitively named methods such as `getDeviceID`, `getStateCode` or `getEntrySize`. Additionally it is no longer possible to read data with the wrong datatype because *DoubleEntry*, *IntegerEntry* and *StringEntry* exclusively provide type-safe getter methods for received values. Practical examples can be found in section 5.4.4, 5.4.5 and 5.4.6.

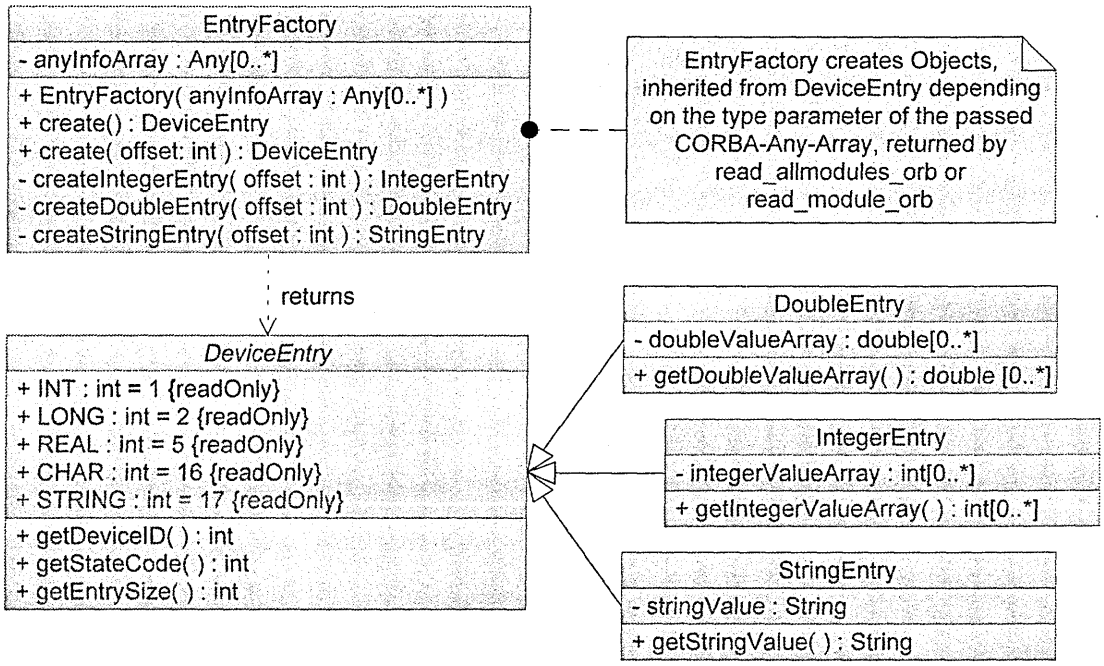


Figure 5.21.: *DeviceEntries and DeviceEntryFactory*

The introduced wrapping mechanism allows the current state and value information for all devices to be acquired using the `getDeviceEntries` method. This is done in a human readable way but so far without delegating data to listeners. The *Caress absdev*-Interface does not provide any callback functionality that can update the client with changes and so

periodical polling of *getDeviceEntries* is the only valid solution to keep the system up to date. Caress distinguishes between slow, normal and fast devices which require a rest period of 10 seconds, one second and below one second between subsequent polls. By reason that the *Caress Client* is no more used in the new design, all polling facilities needs to be reproduced by Open Inspire. For this task the class *DeviceTrigger* (see figure 5.19) provides a timing source that subsequently calls the `getDeviceEntries`-method 9 times for *normal* and one time for *slow* devices with a period of one second. Explicitly reading *fast* devices is not necessary, because reading *normal* devices includes reading *fast* devices and reading *slow* devices includes all.

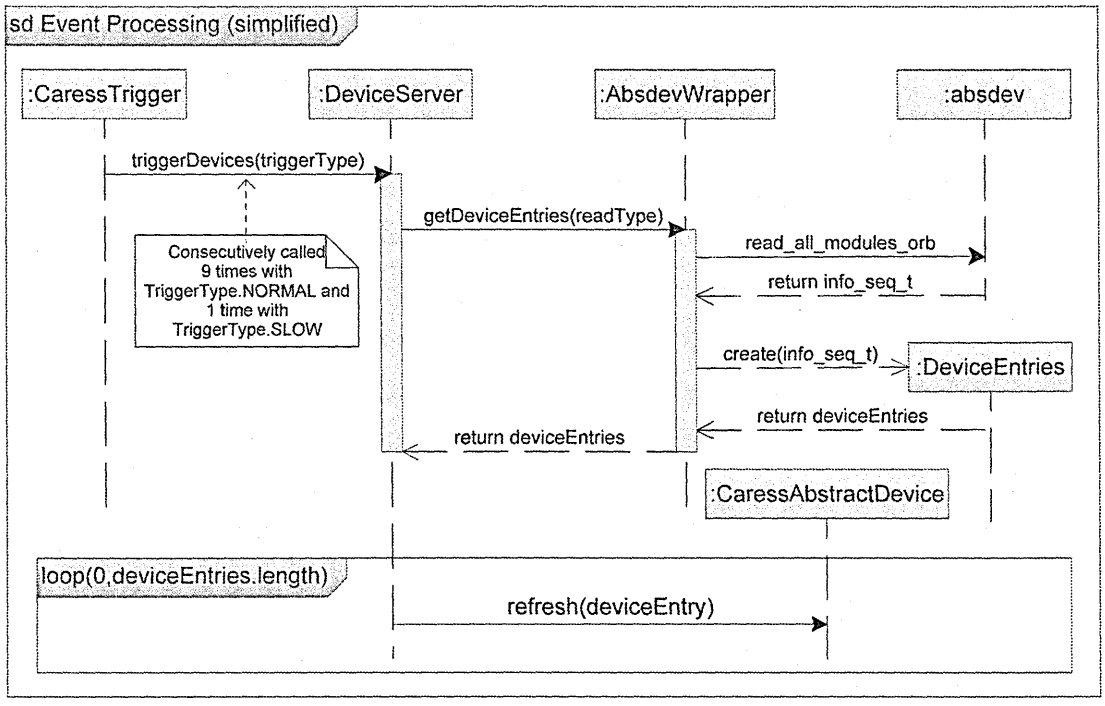


Figure 5.22.: *Polling and the Event Delegation Model*

The last missing part of the architecture is the delegation mechanism that forwards current data to corresponding listeners. Figure 5.22 shows a simplified survey of the event processing architecture. The first step is the periodical call of `getDeviceEntries` method, which returns a *DeviceEntries*-object for the selected device type (normal or slow devices). This *DeviceEntries*-object contains a list of *DeviceEntry* which will be iterated whenever new data is available. During this iteration, the *device ID* of each *DeviceEntry* is compared with all

IDs of the registered *AbstractDevice* entries. Whenever the ID of the *DeviceEntry* matches the ID of the *AbstractCaressDevice*, the abstract method *refresh* of *AbstractCaressDevice* is called and receives the corresponding *DeviceEntry*-parameter. So each implementation of *AbstractCaressDevice* such as *CaressPositioner*, *CaressCounter* or *CaressDetector* automatically receives a device related *DeviceEntry* whenever a device value changes.

This preliminary work decreases complexity and development time for all further implementations of *Caress* devices. The following sections help understanding the introduced models by concrete implementations for all devices that are required to perform a scan.

5.4.4. The Caress Positioner OIM

The *CaressPositioner* is an Open Inspire Module that provides all functionality, required to control caress driven motors and read back encoder values. The next two paragraphs show the *CaressPositioner* from the users and developers point of view.

The Users Perspective

The users view of the *CaressPositioner* is a black box with some configuration parameters and a port that links to the *CaressProxy*, which is connected with the corresponding hardware server. Figure 5.23 shows the entry for the positioner ω_s inside an *OI Assembly*.

```
1 <oim id="OMGS" type="org-openinspire-oim-caress-positioner"
    version="1.3.0-m5">
2   <property name="offset" value="2073.96" />
3   <property name="minimum" value="-200.0" />
4   <property name="maximum" value="200.0" />
5   <property name="tolerance" value="0.01" />
6   <property name="deviceParameter" value="OMGS␣114␣11␣0x00f1c000␣1␣
      4096␣2000␣200␣2␣24␣50␣0␣0␣1␣3000␣1␣10␣0␣0␣0" />
7   <port name="caressProxy" ref="caressProxy"/>
8 </oim>
```

Figure 5.23.: *Initialization of a Caress Positioner OIM using an OI Assembly*

While `offset`, `minimum`, `maximum` and `tolerance` are already known from the *DemoDetector* and configured in the same way, the additional and mandatory property named `deviceParameter` has been introduced for the *CaressPositioner* and configures the remote hardware. If a device has already been used with *Caress*, this initialization value can be

assigned one-to-one from already existing *Caress* configuration files (see Appendix B.3.1 on page 314) or created for new devices following the rules from the *Caress Hardware Guide* [DPSR09].

The Developers Perspective

The developers focus is the internals of the OIM. He profits from the wrapped interfaces and structures that are provided by the *CaressProxy*. Figure 5.24 shows that only two classes are required to implement the whole functionality of a positioner.

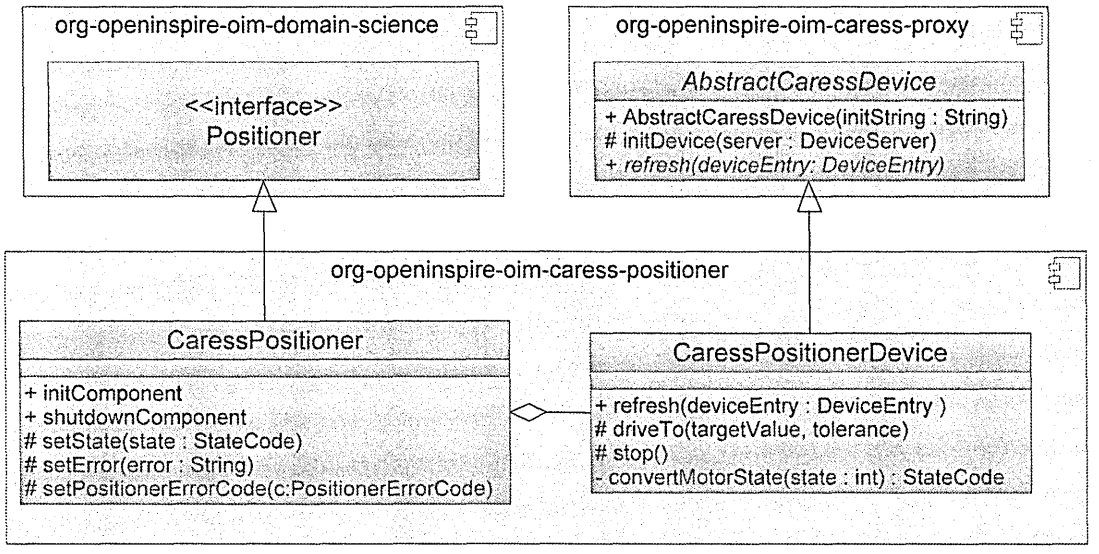


Figure 5.24.: *Survey of the Caress Positioner Classes*

The first class is the *CaressPositioner*, which is a realization of the interface *Positioner* in OIM *org-openinspire-oim-domain-science* (see figure 4.20 on page 126). It is the *OIMs* service class and provides the complete functionality of the *Positioner*-interface to all linked *OIMs*.

The second class *CaressPositionerDevice* is a generalization of the *AbstractCaressDevice* and owned by the *CaressPositioner*. When the positioner is initialized, it creates a *CaressPositionerDevice*-object and registers the object in the device registry of the *CaressProxy*. The `deviceParameter`-property found in the context file is passed to the constructor of *CaressPositionerDevice*, which does all initialization in its super class *AbstractCaressDevice*.

Now the registry takes care that *refresh* is called each time new positioner data is available. In this situation the state and value of the positioner are read from the *DeviceEntry*, stored in a local variables and all listeners are informed about the change. Whenever a new position is available, the value is automatically corrected by means of the offset that has been configured for the OIM.

The other way around is calling operations at the positioner. Beside several *getters* and *setters*, which are used to set local parameters, the two methods `startPositioning` and `stopPositioning` directly call the hardware server through the *CaressProxy*. Here the *CaressPositionerDevice* profits from its generalization of *AbstractCaressDevice* and inherits the preconfigured *AbsdevWrapper*, which already wraps the complex *Caress* calls in a convenient way.

5.4.5. The Caress Counter OIM

The second integral device, required for modeling a complete diffraction scan is the *CaressCounter*. A counter in *Caress* is used for multiple purposes. The main task is the selection of a measurement frame for a detector. In this scenario, the counter is coupled with a detector that simultaneously starts detecting and stops after reaching a predefined setpoint. Here it is important to keep in mind that the hardware of a *Caress* counter is directly connected to the hardware of the detector. With Open Inspire it is in principle possible to combine a *Caress* detector with a software-based counter but it is recommended to always combine a *CaressDetector* and a *CaressCounter* to select an accurate detection frame to the split millisecond. Beside being a timing source, a counter can also represent a counter such as a *Single Detector* or *Neutron Monitor*. This design allows to model measurements where an area detector starts counting until a specific time is reached or a number of neutrons are detected with a monitor counter. It is also possible to use the *Single Detector* instead of an *Area Detector* and count incoming neutrons until a specific timer or the monitor counter reaches its target value.

The Users Perspective

From the users perspective there is no big difference between the configuration of a *CaressCounter* and a *CaressPositioner*. Figure 5.4.5 shows that the module configuration is even less complex since it exclusively requires the already known link to a *CaressProxy* and a *deviceParameter*-property, used to initialize the counter. This example configures a hardware timer and is taken from an *Assembly* at instrument *E7*.

```
1 <oim id="TIM1" type="org-openinspire-oim-caress-counter"
     version="1.3.0-m5">
2    <property name="deviceParameter" value="TIM1␣116␣11␣0x1000000␣2" />
3    <port name="caressProxy" ref="caressProxy" />
4 </oim>
```

Figure 5.25.: *Initialization of a Caress Counter OIM using an OI Assembly*

The Developers Perspective

The developers view of the counter is similar to the positioner with the difference that the service class *CaressCounter* is a realization of the *Counter*-interface of *org-openinspire-oim-domain-science* and that *CaressCounterDevice* does not inherit from *AbstractCaressDevice* but from *AbstractCounterDevice*. Normally it is sufficient that *CaressCounterDevice* directly inherits from *AbstractCounterDevice* but introducing the *CaressCounterDevice* between *AbstractCounterDevice* and *CaressCounterDevice* inside *org-openinspire-oim-caress-proxy* has some major advantages. The *CaressDetector*, which will be introduced in Section 5.4.6 is not only a detector but as well a counter. This means that counter specific functionality is required in *CaressCounterDevice* and *CaressDetectorDevice*. Introducing the superclass *AbstractCounterDevice* removes this redundancy and improves the maintainability.

The second more important reason for this design is an internal dependency between Caress counters. In *Caress* it is not possible to start a counter itself. A counter is always part of a construct called *Acquisition*. Whenever an acquisition is started, one counter needs to be set as master counter and all other counters as slave. Slave counters always act in a passive way and do not stop when a setpoint is reached. They are used to count neutrons, or time, until a master counter reaches its setpoint. With Open Inspire it should be possible to start a counter

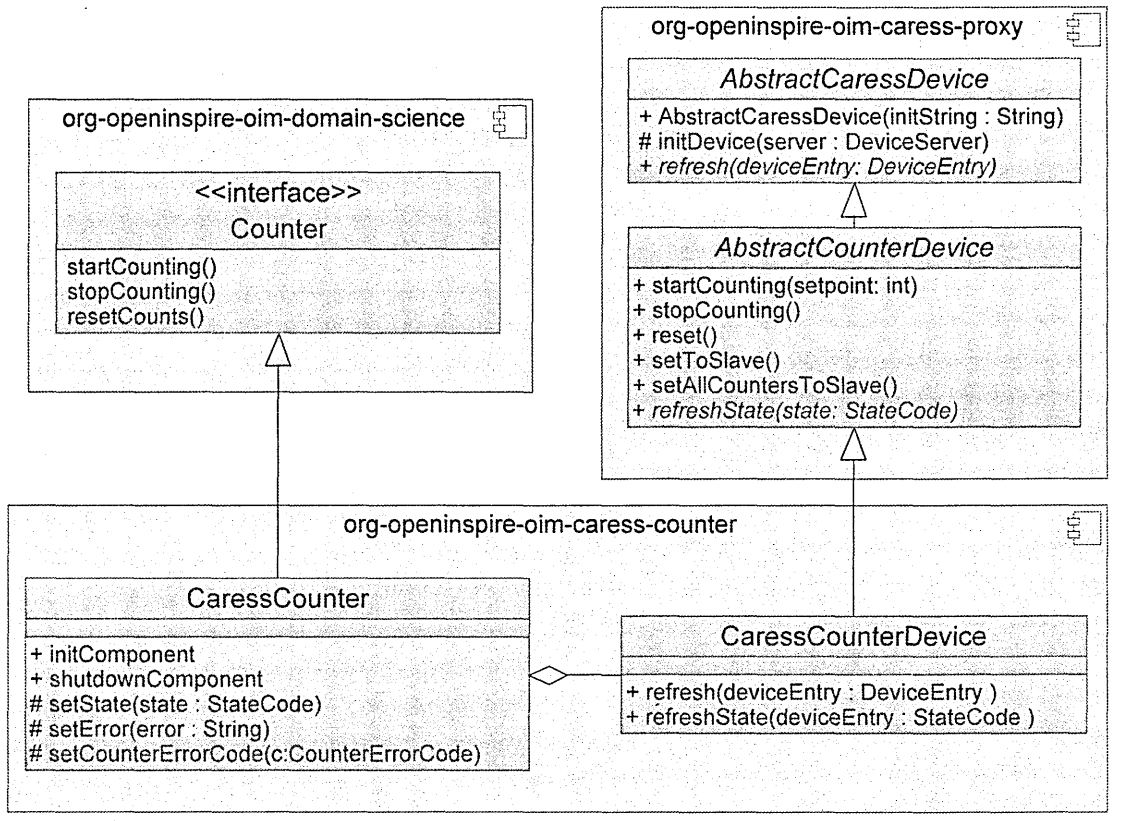


Figure 5.26.: *Survey to the Caress Counter Classes*

without taking care of internal dependencies. To achieve this, every time, `startCounting` is invoked for a counter it is reset and loaded with the value submitted by `setpoint`, all other counters are set to slave state, the aforesaid counter is set to master state and an acquisition is started.

Setting all counters to slave is no trivial task since all counters are independent and do not know other counters. This is volitional to provide a clean object-oriented information hiding. Here the device server solves the problem as it can provide a list of all registered counters, which can be iterated to call the method `setToSlave` for each counter. Having a method in the *DeviceServer* that only returns counters is inflexible and vulnerable because devices could bypass the context and access any OIM. To prevent this issue, the caller needs to pass the class type for the requested objects, and can so only access objects with already known classes. Requesting objects of the class type *CaressCounterDevice* would therefore return

all counters but this would not include the *CaressDetectorDevice* instances, which are also counters. Here the class *AbstractCounterDevice* solves the problem, because asking for all devices that inherit from *AbstractCounterDevice* returns all devices including detectors.

Calling the method `startCounting`, `stopCounting` and `resetCounts` now behave in the same way as any other counter although the underlying design is based on *Caress*' completely different acquisition mechanism.

5.4.6. The Caress Detector OIM

The last missing component, which is required for a neutron scan setup is the *area detector*. While *single detectors* such as counting tubes are modeled as counters, *area detectors* are a new type of device. A detector consist of a neutron sensitive surface which can be accessed as a matrix with two dimensionally arranged channels. The size of the matrix can vary between different instruments and has a size of 256x256 for instrument *E3 / StressSpec* and 128x128 for instrument *E7*. It is a slow *Caress* device which allows the reading of the matrix with a minimum interval of 10 seconds. A *Caress Detector* starts counting when a new acquisition is started and stops when the master counter reaches its target value. This means for Open Inspire that the detection starts and stops with the active counter, which is important to remember whenever a scan is modeled. A detector itself is not only a detector but also a full OI *Counter* that hence also returns the integral value of all detector channels.

The Users Perspective

The configuration of the detector within the OI *Assembly* is nearly identical to the configuration of a counter. Figure 5.27 shows an entry that registers the *area detector ADET* for the instrument *E7*. The only difference between the counter and detector is the `deviceParameter` that initializes a detector with name *ADET* of kind *121*[25] at VME Bus *11* at address *0x000000*, with a low high channel of *0x400* and *128x128* detector channels.

```
1 <oim id="ADET" type="org-openinspire-oim-caress-detector"
      version="1.3.0-m5">
2    <property name="deviceParameter" value="ADET␣121␣11␣0X8000000␣
        0X400␣128␣128"/>
3    <port name="caressProxy" ref="caressProxy"/>
4 </oim>
```

Figure 5.27.: *Initialization of a Caress Detector OIM using an OI Assembly*

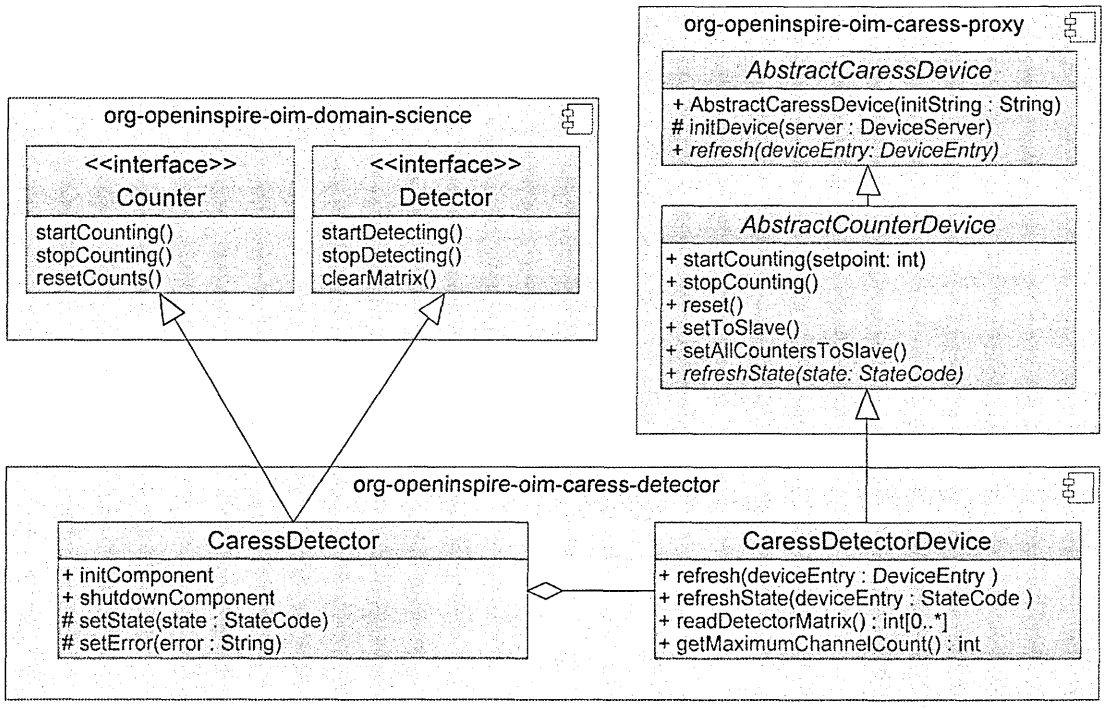


Figure 5.28.: *Survey to the Caress Detector Classes*

The Developers Perspective

The class diagram in figure 5.28 shows that the basic layout of the detector is nearly identical to the counter. The *CaressDetector* primarily realizes the *Detector*-interface but the fact that the detector is also a counter, for the channels integral value, suggests that the detector should be additionally available as a counter. This is done by the realization of the *Counter*-interface. To remove redundancy for the counter implementation and mark the detector as a counter, the *CaressDetectorDevice* inherits from *AbstractCounterDevice* instead of *AbstractCaressDevice*. As written in Section 5.4.5, marking the detector in the device registry as a counter is

[25] Caress Device Kind 121 = HMI X18 Detector

necessary to allow setting the detectors integral counter to slave before starting a *Caress* acquisition.

5.4.7. Summary and Usage

Caress is one of the most comprehensive instrument control systems for neutron scattering experiments world wide. With support for over 130 different device types and a base of about 164.000 lines of code it is expedient to use *Caress* for proving OI's capabilities concerning the reusability of already existing legacy hardware functionality. In this regard it was possible to confirm that *Caress* based hardware can be used with Open Inspire in compliance with all previously defined requirements such as reusability, modularity, minimization of downtime and easy usability.

Reusability and Scalability

The support of *Caress* driven hardware allows scans to be performed with legacy hardware that was intentionally designed not to be controllable by other systems. The requirement that no interfaces or code of the legacy system has to be changed has been fulfilled and allows switching between Open Inspire and the *Caress Control Software* without changing the hardware setup. Users that already operate a *Caress*-controlled instrument and want to use additional third party hardware that is incompatible with *Caress* can therefore prevent downtime and migrate step by step to the new setup or even mix existing *Caress* hardware with new components. The reusability of all existing and working functionality not only saves several man-years of development but also provides access to devices which could otherwise not be reimplemented because of missing code or device documentation.

A different perspective shows the significantly improved system scalability. Open Inspire's architecture allows a modular extension of the instrument capabilities proportionally to the increasing instrument demands. While external components can be simply added to *Caress* driven instruments, it also smoothes the reusability of *Caress* components in scenarios outside of the *Caress* development focus.

Reduction of complexity

Open Inspire successfully hides the complexity of the internal structural and dynamical complexity of *Caress*. Using a *Caress* device such as a positioner, counter or detector requires nothing more than adding the OIM-tag for the corresponding device and setting the device specific parameters. The device configuration is realized by only one `deviceParameter`-String that can be taken one-to-one from a *Caress* configuration file and significantly simplifies the configuration. The issue that *Caress* devices are sometimes spread to different device servers is solved by an additional *Caress Proxy* OIM. Each *Caress* OIM needs to be connected to a *Proxy*-OIM, which creates a link to a device server. This furthermore makes it possible to use multiple *Caress* instances in one *Assembly* at the same time.

The developers perspective is a bit more complex than the users perspective. While the user needs to know how to configure a couple of black boxes, the developer requires a deeper insight into the *Caress* internals. *Caress* is not built in a way that allows the mapping of devices one-to-one to object-oriented Open Inspire Modules due to internal dependencies and hidden assumptions. To improve the maintainability and facilitate the development, all complexity has been shifted into the *Proxy* OIM. The structural and dynamical complexity is therefore wrapped at one position and made available by easy to use wrapper interfaces. Developers that only want to implement new *Caress* devices can therefore benefit from a second level of complexity reduction and need no further knowledge about *Caress* internals. The complete complexity required to map objects and dynamical code is therefore limited to one OIM, which can be maintained by a specialist. At the best this code is never touched, in the worst case of bugs it can be fixed at one place.

Comparison with other systems

Caress is a rather complex example compared to the wrapping of other legacy systems. The interfacing of *Caress* to Open Inspire required much development work such as wrapping non object-oriented functionality by objects, converting state machines and dynamical code to transparent event mechanisms and disentangling hidden dependencies. While other systems such as SICS, TANGO or EPICS often only need a subset of this development work and at

best objects and events need only to be wrapped, *Caress* is a good example that shows that even systems of high complexity can be integrated into the Open Inspire environment.

5.5. Experiment Simulation with McStas

Section 5.4 showed how a user can easily replace devices such as the *DemoPositioners*, *DemoDetectors* and *DemoCounters* by their *Caress* equivalents and thus use real hardware instead of simulated demo devices. For a user it is not more than substituting the corresponding *Device* OIMs in the *Assembly* and reconfiguring the OI *Properties*, while the complexity of the internal implementation remains hidden.

This section shows how the same methodology can be applied to instrument and experiment simulations. With the *Demo Devices*, OIMs are already given that allow to simulate instrument hardware but with the limitation that these devices only provide artificial demo values and are not connected among each other to a physical context.

With the OIMs covered in this chapter it is now possible to run complete *Monte-Carlo*-based simulations that do not only produce random values but incorporate the instrument and experiment physics. Scans thus can be performed with real hardware using the *Caress* OIMs introduced in section 5.4 but also with virtual devices that provide a physical simulation of the environment including the neutron optics, specimen and devices such as detectors.

One use-case is to prepare, create and test a scan setup prior to the real experiment to identify problems, select scan windows or familiarize oneself with the environment. An other advanced use-case is not to just switch between virtual and real devices but to operate them parallely. A scan can hence for instance perform a real measurement and in parallel simulate the next scan step to identify points of interest that should be measured with a higher accuracy.

5.5.1. Realisation Ways

Section 2.4.3 on page 44 introduced the *Monte-Carlo-Method* for the simulation of beamlines as well as common neutron beamline simulation frameworks. From the three most popular frameworks McStas, VITESS and NISP, the first one has been selected to model OI's simulation OIMs. McStas has already been used to simulate the instruments *E3*, *E7* and

StressSpec, runs on all important Operating Systems, comes with a comprehensive repository of components and can be easily coupled with OI through McStas' Command Line Interface.

Writing a new simulation framework was no option because McStas already fulfills all requirements for the simulation framework. Benefitting from a probably better integration in OI is disproportionate to the unnecessary time effort and redundancy.

The following two subsections introduce two different ways to approach the integration of McStas with Open Inspire. The first one separates the physical model of the instrument from the model that represents the hard- and software. The second one uses OI to model both the physical and control system setup and creates the McStas configuration on-the-fly.

Interfacing the McStas Simulation Engine

McStas comes with a couple of tools that allow to run it entirely from the command line. This makes it possible to remote control an installed McStas instance from a Java program and thus from Open Inspire. To do this, an existing McStas instrument description or a new one can be created that contains the physical setup of the instrument. McStas allows to define variables inside this file whose actual values can be passed to the McStas simulation engine. This for instance allows to pass the positions of positioners, the size of a detector or the number of neutron counts to the simulation engine and to create a simulation output for the specific parametrized scan. During the simulation, the results such as the calculated detector matrix are written to a file and can be read back for further processing.

Modeling the complete Experiment with Open Inspire

An other way is to model the complete instrument entirely from OI. For this approach it is possible to make use from McStas' modular configuration blocks inside the instrument description file. For each component type, a corresponding OIM need to be created that comes with the same configuration options, modeled as *OIProperties*. These configuration values are read during the startup of the *OI Assembly* and used to create an *instr*-file. This will be compiled to an executable McStas instrument, which can then be handled in the same way as explained in the previous section.

The advantages compared to the first approach are:

- No external *instr*-file is required since it is created on the fly.
- The complete instrument can be modeled using one configuration file, the *OI Assembly*.
- The *Assembly*-based setup allows to model a McStas instrument in a graphical way.
- It is possible to configure a McStas setup without knowing how to write an *instr*-file.
- The parameters and their validity can already be checked by the McStas OIMs so that misconfigurations, typos or syntax errors can be prevented and lead to fail-safety.

The disadvantages are:

- It makes only sense when a new experiment is created. For already existing simulations it is easier to reuse the existing *instr*-file than recreating a new setup.
- Existing *instr*-files can often not directly be converted to a setup that mirrors the device-based setup of the instrument. While the existing *E3* simulation description directly reflects the real structure of the instrument by modeling the axes as component blocks, the *ENGINX* setup interweaves the device parameters with the code.
- The *instr*-file need to be recreated every time when the simulation is started. This slows down the simulation but can be solved by cashing the setup and recompiling it only in the situation of changes.
- It is a high effort to create OIM equivalents for all *McStas* components. This can be solved by a conversion algorithm that parses the McStas component interfaces and creates corresponding OIMs that mirror the parameters as *OIProperties*.

To map the McStas component design entirely to Open Inspire Modules, another problem needs to be solved. Figure 5.29 shows a snippet of the *E3 instr*-file (see Appendix B.4.1 on page 316), which contains the definition of the positioner *xT*.

```
1 COMPONENT xt_motor = Arm() AT (XT, 0, 0) RELATIVE zt_motor ROTATED
    (0, 0, 0) RELATIVE zt_motor
```

Figure 5.29.: *Definition of the McStas COMPONENT xT*

Here is clearly seen that the axis is not only modeled as a McStas component from type *Arm()* but also comes with geometrical coordinates for the position and rotation in a three-dimensional room. A component can have an absolute position, which is indicated by the keyword *ABSOLUTE* and a preceding coordinate or have a relative position to an other component such as the example's positioner *zt_motor*. In the second case it is tagged with the keyword *RELATIVE* and three values that describe the distance to the object it is relative to. To describe the rotation of an object it is beyond that possible to describe the rotation of the object using the keyword *ROTATED*. It is as well configured through three values for the three rotation axes and can in the same way be *ABSOLUTE* or *RELATIVE* to the rotation of an other object.

OIMs have no geometry informations by default, but these are necessary to describe the position of a McStas component when the *instr*-file should be created from the *Assembly*. One solution would be to add coordinates to each device but this has the drawback that the OIMs become overloaded with meta information when it is not necessary and that the OIMs become coupled to the specific McStas coordinate model. This is why it is more meaningful to create a separate *Geometry* OIM that holds the information about the position and rotation. When a position or rotation is relative, it can simply be linked to another object using an *OIProperty*. To create the relation between Geometry- and *Device*-OIMs, a *Geometry* OIM possesses an additional *OIPort* that can hold a reference to the OIM whose position shall be described. When the McStas simulation file is created, it is now possible to search for all *Geometry* OIMs in the *Assembly*, query the linked *Device* OIM and create the corresponding McStas component blocks. Each *Geometry* OIM can be queried for an absolute position. When the OIM is relative, all position and rotation offsets are accumulated by following the links to the next OIM with absolute position or rotation. The usage of *Geometry* OIMs is not only useful for the creation of McStas setups but can also be used to model the instrument geometry in a three-dimensional. For instance to view the setup by means of a CAD program.

5.5.2. McStas Components

For all instruments covered in this thesis, an existing McStas *instr*-file is already available. This and the fact that the *ENGIN-X* McStas model cannot be simply converted to an *Assem-*

bly-based setup[26] make it meaningful to implement the McStas OIMs using the first approach where an existing instrument file is loaded instead of generating it on the fly.

To make the *Caress* or *Demo* OIMs directly replaceable by McStas OIMs, the three OIMs *McStasPositioner*, *McStasCounter* and *McStasDetector* are available and derive from the corresponding interfaces from *domain-science*. Beside these OIMs, an additional OIM called *McStasProxy* handles the communication between the McStas OIMs and *McStas*. This concept is already known from the *CaressProxy* (section 5.4.3 on page 213). Since the instrument geometry is already defined inside the McStas *instr*-file, no *Geometry* OIMs are required.

McStas Positioner

Since the *McStasPositioner* derives from the *Positioner* in *domain-science* it has the same base *OIPropertes* and *OIPorts* such as the *CaressPositioner* or *DemoPositioner*. The only difference in the configuration is the additional reference to the *McStasProxy* OIM, which queries the *McStasPositioner* for setpoints and updates its current position. While the *CaressPositioner* has a delay between setting and reaching a setpoint due to concrete hardware that needs to be driven and the *DemoPositioner* has a delay because an internal thread simulated the positioning, the *McStasPositioner* immediately reached the given setpoint. It is a container for the setpoint value that is queried from the *McStas* OIM and not a simulation of the positioners interness.

McStas Counter

Similar to the *McStasPositioner*, the *McStas* counter only holds the value for a neutron count that should be reached during the experiment and is passed to the *McStas Simulation Engine*. But this time, the count value does not directly reach the setpoint but is updated corresponding to the simulation state so that the actual simulated neutron count can be read. It is up to the simulation setup if a neutron counter or timer is used.

[26] Due to the already covered problem that the parameters are interwoven with the *McStas* code.

McStas Detector

The *McStasDetector* is a facade to the actual detector values of the experiment. Is is automatically updated by the *McStasProxy* when the matrix of the simulated 2D-detector changes. Due to a periodical update it always holds the current detector matrix subject to the current update interval.

McStas Parameter

The *McStasParameter* is an additional OIM that allows to pass arbitrary *String* values to McStas. Until OIMs are available that reflect all possible McStas components it is with it possible to pass additional values such as a temperature or the state of a switch.

McStas Proxy

The *McStasProxy* finally handles the communication between OI and McStas. Appendix B.1.1 (page 309) contains an example *Assembly* for a McStas simulation on *E3* and appendix B.4.1 (page 316) the corresponding McStas *instr*-file.

Configuring the *McStasProxy* is not more than setting three *OIProperties*. A path to a folder where the simulation files are stored, the name of the *instr*-file and the name of the simulation output. To interlink the McStas OIMs with the *McStasProxy*, all McStas OIMs have a port named `mcStasProxy`, which accept a reference to a *McStasProxy*. When the *McStasProxy* is started, it reads and validates the corresponding *instr*-file and checks if the input parameters of the scan correlate with the ids of the *McStasPositioner*, *McStasDetectors* and *McStasCounters*. If the file is valid, OI calls the McStas compiler to create an executable simulation.

5.5.3. Scanning with McStas

The scan engines (section 5.3 on page 195) that have already been used with the *Caress*- and *Demo*-OIMs, can be used in the same way with the *McStas* OIMs. The setup is identical with the one in figure 5.7 (page 196) except from the device OIMs, which are substituted by the McStas equivalents.

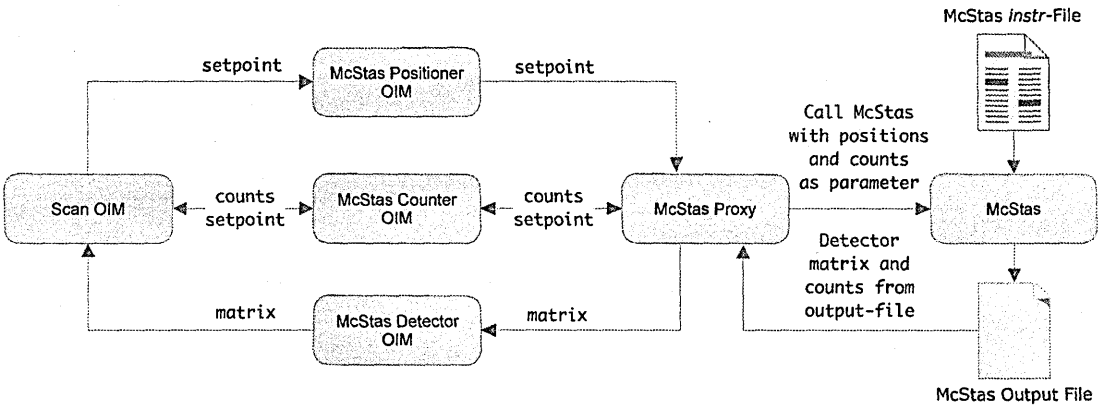


Figure 5.30.: *McStas Communication Diagram*

Figure 5.30 shows the internal dataflow between the McStas OIMs, a *Scan* OIM and McStas itself. While step *Initialization*, *Snapshot* and *Long-term Storage* of the demonstration scan (figure 5.8 on page 198) are identical for all *Device* OIMs, the internal realization of the data flow during the positioning and detection makes the difference.

POSITIONING: When the scan reaches the *Positioning* step, the setpoints are passed to the *McStasPositioner* OIMs but since no real hardware is driven, the current position values immediately follow the setpoints. The *Sync.byEvent* command consequently directly unblocks so that the detection step follows without delay.

DETECTION: Prior to the detection, the time or number of neutrons used to limit the detection period are set as an *OIProperty* of a *McStasCounter* OIM. Starting the detection by calling `startDetecting` and `startCounting` now triggers the *McStasProxy* to read the setpoints from the *McStasCounter* and *McStasPositioner* OIMs. These values are passed as parameters to the simulation executable, which has been created from the *instr*-file.

To allow Open Inspire getting the simulation state while the simulation is running, the McStas component `Progress_bar` needs to be added to the *instr*-file[27]. This component periodically writes the current count and detector values to a file. Otherwise the simulation runs in background and OI only gets a feedback when the simulation is complete. The *McStasProxy* automatically detects when the output files change, reads the current values and passes them

[27] Appendix B.4.1 on page 316 shows the usage of the McStas `Progress_bar` component.

to the *McStasCounter* and *McStasDetector* OIMs. These in turn notify listeners such as GUI widgets or a *Scan* OIM[28].

OI comes with the three sample *Assemblies E3-McStas*, *ENGINX-McStas* and *Vanadium Example*. To run these *Assemblies* it is necessary to make sure the the *instr*-files are available at the path given as property of the *McStasProxy* and that McStas is installed and made available through the system path.

5.5.4. Limitations, Optimizations and Outlook

The fact that original McStas *instr*-files can be used to model an experiment and the ability to pass arbitrary parameters makes it possible to control nearly any *McStas* experiment with Open Inspire. Unfortunately a technical limitation of the McStas `Progress_bar` component hinders OI to get status updates until 30% of the simulation is already progressed. This however has no effect on the result of the experiment but prevents the UI to update values during the first part of each instrument step.

McStas normally counts until a fixed number of neutrons have been processed but not until a number of neutrons reach an independent detector such as a monitor or until a specific time is reached. To get results for such scenarios it is possible to split the simulation process in a large number of short measurement steps. After each step the results are compared and checked if the target value has been reached. If this is true, the simulation stops and all values are accumulated and returned to the *McStasProxy*. Otherwise the simulation continues until the values or a measurement time is reached.

If a continuous texture analysis shall be performed with the McStas OIMs, it is not possible to manipulate the positioner axes while the simulation is running. This makes the result of the experiment with *Caress* differ from the results with McStas. The solution here is similar to the solution in the last paragraph by splitting the simulation into short measurement steps, driving the axes between these substeps and accumulating the measurement results. The drawback is a significantly reduced speed of the overall simulation process due to the larger number of individual simulation steps.

[28] For instance to refresh a widget that displays the detector matrix or to inform the *Scan* OIM that the current scan step has completed.

5.6. Autonomous Instrument Alignment

The final example demonstrates Open Inspire's strength in combining various technologies. It integrates a legacy system as well as new hardware, OIMs that perform mathematical operations, OI's remote service infrastructure and uses a specialized User Interface Client.

The function of the setup is an automatic alignment of a specimen on a diffractometer's goniometer table as well as the adjustment of slits that allow limiting the neutron beam to a defined shape and position where it shall intersect the sample.

5.6.1. The classical calibration process

Figure 2.2 on page 26 shows a neutron diffractometer with a specimen (red crystal) on the goniometer table (figure 2.4 on page 28). Positioning this specimen exactly in the middle of the table is typically done manually by means of an optical levelling system (dumpy level). This alignment is time consuming because various measurements and calculations need to be performed so that the specimen remains in the center even when the ω axis (horizonal rotation) or the axes ϕ (vertical rotation) and χ (tilt) of the eulerian cradle (figure 2.7 on page 29) are driven.

When the sample is centered at the table, the neutron beam can pass through it, but as long as the beam comes directly from the neutron source without adjusting its size and shape, the scattering results are inaccurate[29]. This problem can be solved with help of a cadmium[30] slit (figure 2.3 on page 27) between the source and the specimen. This slit constrains the beam usually to a cross-section of 1x1 or 2x2 mm^2 and can be adjusted using the three positioning axes *PST* (translation), *PSR* (rotation) and *PSD* (distance). Aligning theses axes is done by measuring the neutrons that pass the specimen while varying the slit axes until the majority of neutrons hit the sample at the desired position. For the neutron detection a single detector (2.6 on page 29) is mounted behind the specimen and for the selection of the desired sample position, a second slit is mounted at the sample position.

[29]The reason is the irregular shape of the neutron beam and its ambiguous intersection with the sample.
[30]Cadmium is a good neutron absorber and blocks neutrons from passing the slit around the opening.

5.6.2. The Automated Alignment - Hardware Setup

While a manual instrument calibration 1) takes a considerable amount of time and 2) hardly reaches the same accuracy as a machine-driven alignment, an automated alignment solves the two stated problems and beyond that makes the instrument calibration reproducible. For such an automated alignment hardware enhancements have been applied to the instruments and extensively been covered within the scope of the diploma thesis [Flc05].

Optical Position Recognition

The first extension is an optical system that is pannable to the goniometer table and grabs the position of a sample. Figure 5.31 shows that the system consists of a camera that is directed towards the sample and a light source that illuminates the sample from the back.

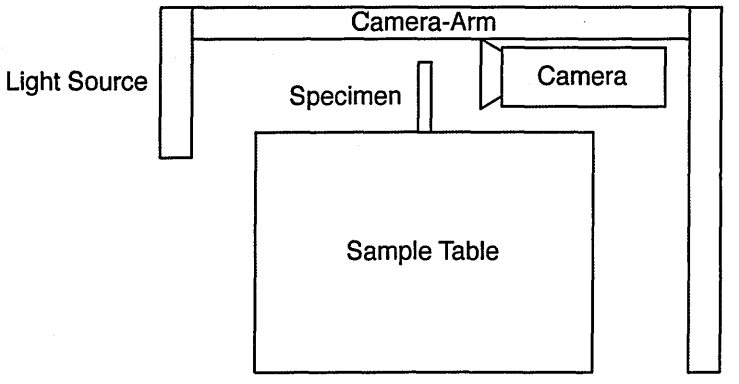


Figure 5.31.: *Calibration - Camera Setup*

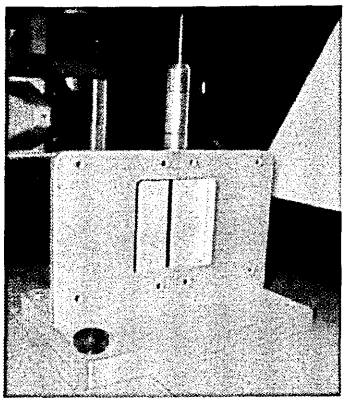


Figure 5.32.: *Pin and Slit*

To gain a high recognition accuracy of the sample position, the whole setup has been optimized for a high-definition edge detection. The active optics thus consists of a monochrome camera with a resolution of 1280x1024 pixels and sensibility of 380 to 1000 nm. To avoid deformations during the variation of the sample distance, a telecentric objective with a reproduction scale of 1:1 is mounted in front of the 2,3" Progressive Scan CCD sensor.

To process the data, which is transferred to a PC using a *Siso Microenable III* frame grabber card, the image processing system Common Vision Blox [cvb12] is used. The system is only available for *Windows*, which makes it necessary to separate the camera processing system from the *Linux*-based instrument control components. Software details follow in section 5.6.3.

Calibration Standard Sample

The second extension is a calibration sample, which consists of a *pin*, *powder-tube* and *slit*. It is shown in figure 5.32 and can be mounted centrally on the goniometer table. Driving the slit in the neutron beam effects that neutrons which do not pass the central window are caught by the sidewise cadmium masks. Driving the sample in height of the pin or powder tube into the beam allows to create a predefined scattering result. And driving the pin in the camera's range of vision allows to accurately detect the position of the sample on the goniometer.

Primary Slit Automation

The third extension finally equips the three axes of the primary slit *PST*, *PSR* and *PSD* with motors and encoders. These allow to automatically drive the axes and read back their positions and are an important prerequisite for the automatic calibration of the beam-shape and position.

5.6.3. The Automated Alignment - Software Setup

Figure 5.33 shows an OI *Assembly* for a calibration setup. It includes controller OIMs for three calibration steps (orange), hardware proxies (blue), the *ServiceRegistry* OIM (grey), known from section 4.4.1 (page 141) and two helper OIMs (grey) that provide mathematical fitting and general calibration routines, which are shared between all calibration OIMs.

The Caress Devices

The *Assembly* contains proxies for the goniometer positioners xT, yT, zT and *omgs* as well as the primary slit axes *PST*, *PSR* and *PSD*. The detection is carried out by monitor-counter MON, which detects all neutrons that enter the experiment environment, and counter-tube SDET, which detects the neutrons that leave the experiment in beam direction (see paragraph 2.3.1 on page 29).

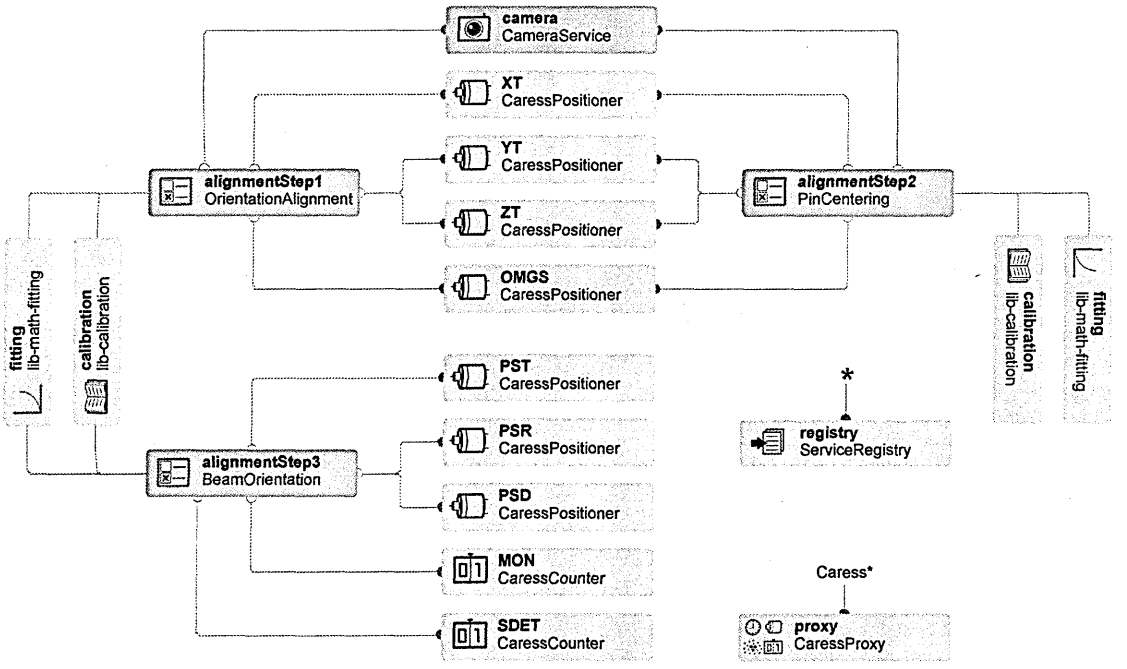


Figure 5.33.: *Calibration Assembly*

The Camera Service

The *CameraService* OIM acts as a device proxy to the *Optical Position Recognition System*, which allows to obtain access to the camera picture and edge information. The camera system is restricted to run an *Windows* systems so that it has been decoupled from the OI-Server using a web service. This does not only prevent OI in loosing its platform independency, it also allows that multiple devices can have read access to the camera server at the same time.

The camera server *OI CamServer* can be downloaded from the OI store[31]. It is realized in the form of a *C++* application and uses the library Common Vision Blox [cvb12] from *Stemmer Imaging* to receive and process the image data. Beside a server-side GUI that displays the image and edge informations, a SOAP based webservice has been introduced to publish the information over a network connection. OI is thus able to receive the current camera image and to remotely access the image processing capabilities of CVB. Figure 5.35 shows the values of the goniometer axes and a live view of the camera as part of the OI *Monitoring Client*. The silhouette projection (page 26 of [BK04]) of the calibration pin allows to clearly differentiate

[31]The *OI CamServer* in its current version 1.23 can be found in the sources section of the OI webportal.

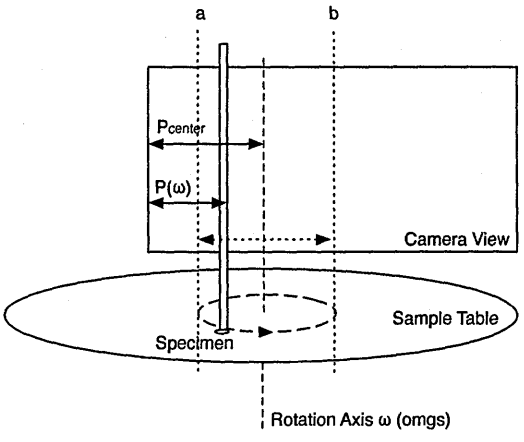


Figure 5.34.: *Optical Alignment - Side-face*

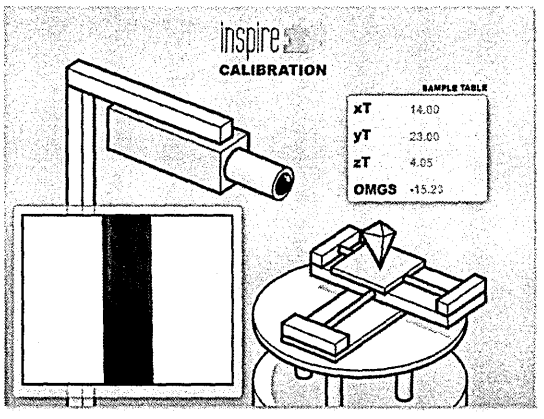


Figure 5.35.: *Camera Monitoring Client*

between the dark pixels of the pin and the light background. Considering the edges of the projection it is possible to correlate the detected[32] edges of the pin with its absolute position on the goniometer table. Figure 5.34 shows that an uncentered pin on the goniometer results in horizontally oscillating movement of the grey bar in the image.

Fitting- and Calibration Routines

Since it is not possible to optically evaluate the exact positions in a three-dimensional way, a mathematical model is necessary that establishes the relationship between the 2D-camera image, the relative values of the *Positioner* axes and the absolute position of the *calibration sample*. For this purpose, a model function is built that links the edge locations of the image to the values of the positioner axes. After collecting a sufficient number of measured edge- and its respective *Positioner*-values, it is hence possible to determine the unknown variables of the model function using a fitting algorithm.

The fact that such a fitting algorithm is predestinated to be reused in other use-cases, such as the slit alignment, makes it meaningful to implement it in a general way, which allows to share it between different scenarios. OI hence comes with the OIM *org-openinspire-oim-lib-math-fitting*, which provides an optimized *Levenberg-Marquard*-based least-square fitting algorithm. The algorithm has been ported from *Numerical Recipes in C* [FPTV92] to *Java*,

[32] OI allows three *edge-detection* mechanisms based on the threshold, contrast and 2nd derivation, which are described in more details on page 68 of [Fle05].

optimized and equipped with a customizable interface. Each OIM that requires to use the algorithm can add it as OIM dependency and use it with an arbitrary model function.

Several interfaces and algorithms, which would be redundant in all calibration steps but uninteresting outside the context of an instrument alignment have been added to the OIM *org-openinspire-oim-lib-calibration*. This is added as dependency for all *alignment steps*.

5.6.4. The Alignment Steps

Figure 5.33 shows that each step of the alignment process comes in form of an independent OIMs. Since they all inherit from the interface *StateControlFeedback* (*domain-science*), they come with `start-`, `stop` - methods and a registry that allows to register listeners for status messages. This allows them to be controlled with the same *StateControlFeedback*-widgets used for all other scans. Customized widgets that bind against the interfaces of the individual steps are as well available and visible in screenshot 5.40 at the end of the chapter.

The following sections give a brief survey of the calibration steps[33]:

Step 1: Orientation Alignment

Performing an *Orientation Alignment* is a precondition for all other calibration steps and is thus always the first step. It identifies the orientation of the translation axes in relation to the camera picture, because the manifold number of *positioner* axes are indeed mounted at defined positions but their global orientations and correlations vary between the instruments. Reasons are different hard- and software settings, differences in the setup of the motors and gears as well as varying encoder orientations. Driving *omgs* in positive direction can thus result in a clockwise- but as well in a counter-clockwise rotation and the global direction of a translation axis such as xT not only depends on the setup but also on the current rotation of the goniometer table[34]

The *Orientation Alignment* provides an automatic evaluation of the axes parameters so that the subsequent calibration steps are applicable on all instruments, independent from their

[33] More details to the mathematical backgrounds can be found in [Fle05] on page 134 to page 140.
[34] Figure 2.2 (page 26) shows that xT and yT are both mounted on the goniometer and can thus be rotated using the table rotation axis *omgs*. Driving omgs 180° e.g. inverts their directions in the video image.

setups. Figure 5.33 shows that the *OrientationAlignment* uses the four *Caress* positioners xT, yT, zT and *omgs*, the *CameraService* as well as the mathematical *Fitting* OIM.

Prerequisite for the orientation determination is that the calibration sample is mounted on the goniometer table and that its pin is visible in the camera image. The *Orientation Alignment* now drives the pin for 12 equidistant distributed ω-positions over a measurement range of 180° to different x and y positions and optically determines the horizontal center of the pin. With these measurement values it is possible to calculate the scale-factors, offsets and the horizontal alignment of the axes in relation to the camera orientation.

$$P(\omega, xT, yT) = P_{center} + F_x(xT + x_0)cos(\omega + \omega_0) + F_y(yT + y_0)sin(\omega + \omega_0) \tag{5.1}$$

Equation 5.1 shows the model function, which is fitted using the *Levenberg-Marquard* (*org-openinspire-oim-lib-math-fitting*). P describes the optical edge position of the pin center as a function of the *Positioner* axes ω, xT and yT. The refinable parameters are the linear offsets x_0, y_0 and ω_0, the scale factors F_x and F_y of the translation axes and the center of the goniometer table P_{center} in camera coordinates. With 16 measurement positions, it is already possible to position the pin with an accuracy of $\approx$ 10 μm on the goniometer table's center.

Step 2: Pin Centering

If the results of the *Orientation Alignment* steps are already available, it is possible to use the information about the relation between camera pixels and the translation axes as well as the scale factors to perform a simplified centering of a cylindrical sample. Responsible for the simplified centering is the OIM *org-openinspire-oim-calibration-pinCentering*. It makes use of the same hardware setup, which has already been introduced in *Step 1 : Orientation Alignment* but differs in its internal software algorithm.

$$P(\omega) = P_{center} + A\sin(\omega + \phi) \tag{5.2}$$

This algorithm rotates only ω 180° and collects edge data for 36 ω-positions. If the sample is not centered, the edges sinusoidally oscillate around the center. This behavior can be

described by equation 5.2. Fitting this function results in Amplitude A, which represents the maximal deflection of the cylinder's center P from the goniometer table's center P_{center} and the phase ϕ, which contains the information about the compensatable motion of xT and yT. More detailed information about the mathematics and used algorithms can be found in thesis [Fle05] and the *org-openinspire-oim-calibration-pinCentering* OIM.

Step 2: Primary Slit Alignment

The third alignment step is subdivided in three sub steps and made available through the OIM *org-openinspire-oim-calibration-beamOrientation*. It automatically aligns the primary slit (see figure 2.3 on page 27) so that the neutron beam intersects the specimen with a defined shape at a defined position.

To achieve it, it is at first necessary to determine the direction of the beam. The calibration sample (figure 5.32) is for this purpose mounted at the previously determined center of the goniometer table. However, this time the axis zT is positioned in such a way that the neutron beam hits the slit of the calibration sample. Now ω is rotated in 72 steps around an angle of 180° and the neutron intensity at single detector SDET is measured.

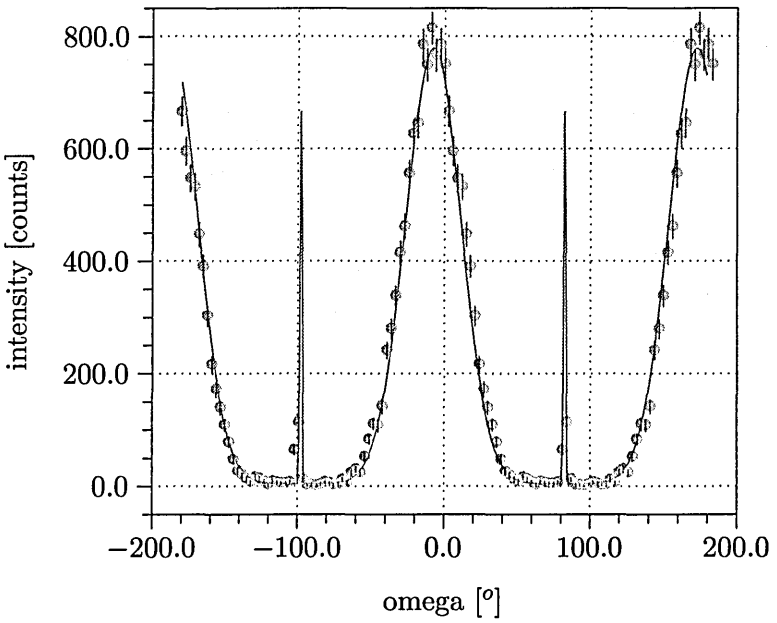


Figure 5.36.: *SDET Intensity versus Omega (SLIT Alignment)*

Figure 5.36 shows the result of the measurement in the full ω range of 360°. The wide maximum at $\sim 0°$ show the position where the beam passes the center of the slit while the ω-positions *-100°* and *80°* show artifacts that occur when the slit is aligned in parallel to the beam. The function can approximatively be expressed with help of the equation 5.3.

$$I(\omega) = A_0 \exp \frac{-(\omega - \omega_0)^2}{2{\Gamma_0}^2} + A_1 \exp \frac{-(\omega - \omega_0 + 90°)^2}{2{\Gamma_1}^2} - A_1 \exp \frac{-(\omega - \omega_0 - 90°)^2}{2{\Gamma_1}^2} \quad (5.3)$$

Figure 5.37 shows the setup with the calibration sample in the center of the goniometer table. Due to the size of the beam it is not necessary that the center of the beam exactly hits the center of the slit. To refine the results and to get the exact position of the maximum, it is now possible to drive the sample in direction of the single detector SDET and to continue the measurement as shown in figure 5.38. With the information of the first scan it is now sufficient to rotate the slit with a smaller angular interval of about *10°*.

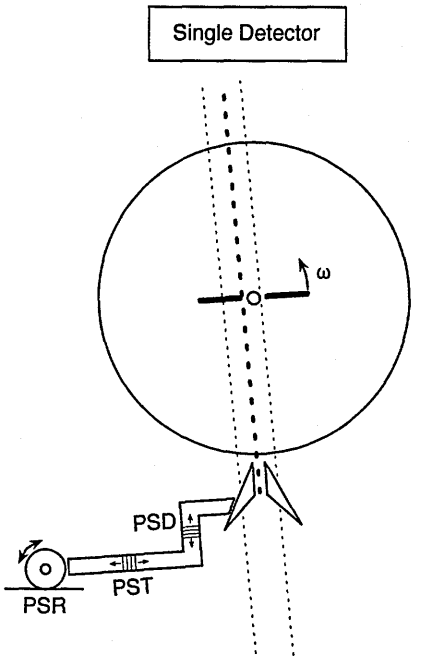


Figure 5.37.: *Primary Slit Alignment 1*

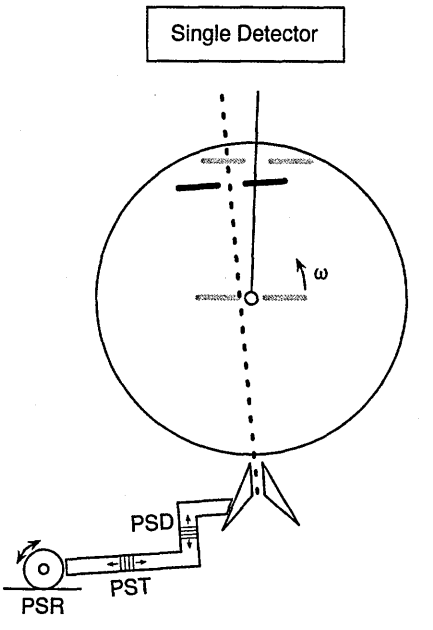


Figure 5.38.: *Primary Slit Alignment 2*

With the calibration sample at the position where the beam shall hit the specimen it is now possible to align the primary slit. For this purpose, the aperture seen in the bottom left corner

of photo 2.3 on page 27 is moved into the beam to restrict it to the size of the apertures opening. Figure 5.39 shows that the slit can be moved forward and backward using the axis Primary Slit Distance (PSD) so that it can be positioned directly at the position where the beam leaves the beam-tube and a position directly in front of the sample.

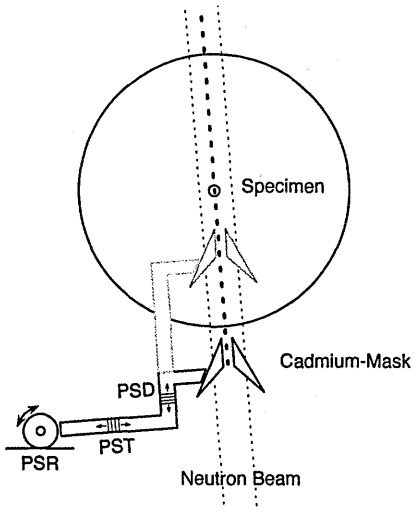


Figure 5.39.: *Calibration - Beam Orientation*

For these both PSD positions, the axis Primary Slit Translation (PST) is moved transversally to the neutron beam. The single detector SDET is now used to measure the neutron intensity for several PST positions analogous to the previous measurement with the slit of the calibration sample. The resulting gaussian curves can subsequently be fitted to obtain the PST positions where the neutron maximums occur. With these two positions it is possible to geometrically calculate the beam angle and to respectively rotate the axis Primary Slit Rotation (PSR). The alignment of the PST axis and the PSR axis can now be repeated until the primary slit construction is aligned with the desired accuracy.

The beam is now restricted to the shape of the primary slit and intersects the sample at the favored position. Due to the previous centering it is now possible to horizontally rotate the specimen using the *omgs* axis and to vertically rotate and tilt it using the eulerian cradle's *phi* and *chi* axes while the intersection with the beam remains in the center of the sample.

5.6.5. The User Interface

Figure 5.33 on page 247 shows that the *RegistryService* `registry` is used to publish the service interfaces of all OIMs over the network. Figure 5.40 shows a GUI that acts as remote client for the calibration process. It can be connected to the remote registry and provides widgets that show the positioner information, a live camera view and a wizard for the configuration and execution of the calibration steps.

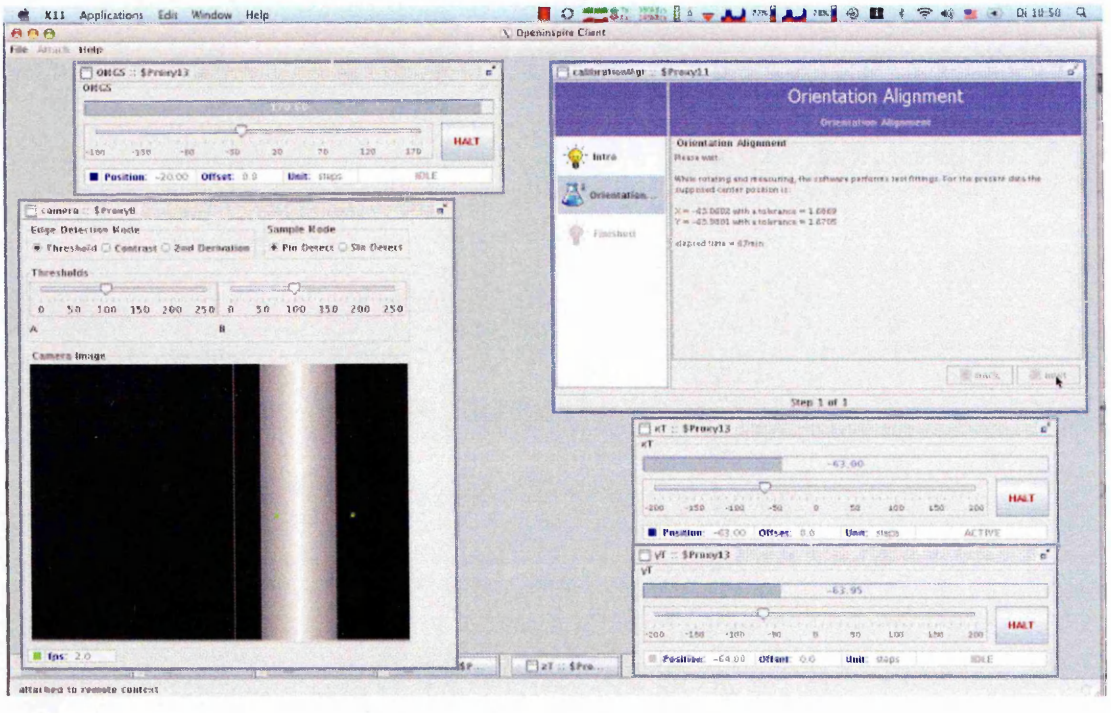


Figure 5.40.: *Calibration GUI Client*

5.6.6. Usage and Limitations

The setup is either partially or completely in use at the instruments *E3* and *E7* (HZB in Berlin), at *StressSpec* (FRM-II Garching), at NECSA (South Africa) and ANSTO (Australia).

As quoted on page 249, the setup allows to position a specimen with an accuracy of 10 μm (*E3*, *E7*, *StressSpec*). Depending on the user requirements and nature of the experiment, a higher resolution is reachable by decreasing the positioner tolerances, which however increases the calibration duration, since more positioning steps are required. The overall accuracy is limited by the accuracy of the positioners, encoders and the camera resolution.

CHAPTER

SIX

SUMMARY, CONCLUSION AND OUTLOOK

6.1. Summary

The PhD thesis has demonstrated that it is possible to answer the research question by introducing a new software and collaboration concept that was proved by the realization of the novel instrument control and data acquisition system Open Inspire.

Chapter 2 contains the current most complete review of instrument software for neutron diffraction experiments. The concepts learned from these systems, interviews with instrument responsibles and the evaluation of contemporary best practices in software design motivated the creation of the Software Requirements Specification found in chapter 3. As a starting point for everyone who wants to develop an instrument system it contains a stakeholder analysis, summarizes the goals and project constraints and a comprehensive list of functional and non-functional requirements. Chapter 4 introduces the reference implementation of a novel instrument system, which meets all stated functional requirements and more than 99% of the ~115 *non-functional* requirements. The system named Open Inspire builds on a flexible and extensible Inversion of Control-based container architecture and comes with a cloud-based collaboration infrastructure to tackle the requirements on a system that reduces redundancy by sharing common functionality. Particular challenges such as meeting requirement [☒ 3.1.7],

which demands that the system combines the power of a monolithic system with the flexibility of a modular system, have been tackled through OI's Inversion of Control-based container architecture. The time constraints of requirement [☒ 2.3.2], [☒ 2.3.3], [☒ 2.3.4], [☒ 2.3.5] and [☒ 2.3.6] have been verified with help of students. Only the documentation requirement [☒ 2.3.9][1] and the automatic continuation of experiments after crashes [☒ 3.4.3][2] have been shifted to a later release. Chapter 5 demonstrates, that the system design and implementation is usable in real instrument environments. All requested features such as a flexible UI, the ability to store measurement data in the *NeXus* format, the execution of scans, the compatibility with legacy hardware, the simulation of experiments and the ability to quickly implement completely new features have been achieved. New functionality can be developed by the community, published in the form of OIMs using the OI store and immediately be used by everyone who operates an OI server or wants to use the OIM as library in its own development.

6.2. Conclusion

This conclusion section briefly summarizes the advantages resulting from the work of the thesis. Instead of an evaluation of the individual chapters, a form has been chosen that illustrates the impact of the work on the stakeholders (p. 58) that may be confronted with the system.

6.2.1. The Outcome for Research Facilities

The first directly affected group form the *Research Facilities* or establishments that operate the instrument system. They already profit from the summary of instrument systems in chapter 2, which facilitates the decision if an existing system shall be introduced.

If Open Inspire is to be introduced as a new system, the SRS in chapter 3 can be used to verify if the operators desired requirements match the requirements met by Open Inspire. If so, they can browse the OI store to identify the modules that are required for the experiments that shall be performed at the instruments. If functionality is not yet provided by available OIMs, such as hardware support for proprietary in-house devices, it can be commissioned

[1] The documentation will be extracted from the thesis when it has been published.
[2] [☒ 3.4.3] requires instructing OIM developers to integrate mechanisms that save the OIM state after a crash.

for development. The advantage is that only necessary functionality needs to be developed and that the development can be split into small OIM-based working packages. This allows a parallel development, a better distribution of the work and results in a faster and more focused development process. Publishing the modules over the store lets other institutes benefit from new developments, so that a natural symbioses emerges, which again lowers expenses and prevents development redundancy. If multiple institutes use the same modules it is furthermore presumable that the functionality is liable to more tests and corporate enhancements. Updates and bug fixes are centrally available and automatically installed by the OI container when they are requested, which simplifies the overall update-process. Open Inspire allows the mixing of modules and operating of new and legacy hardware in parallel, which in turn, allows the smooth application of upgrades step-by-step, rather than requiring the shutting down of the instrument for the replacement of the whole instrument software. It is the only instrument system that provides such a level of collaboration and sharing of functionality while being *Open Source* and free for everyone to inspect the code and making sure that the code satisfies all security and quality interests.

Institutes usually plan to use instrument software for a large period of time such as 10 or 20 years. Open Inspire does not need a global maintainer and even if the development of the core project is stopped, it has no effect on the further process, because OI modules are decoupled from the core and as long the community provides new features, the development can continue. To be independent from the global OI store it is furthermore possible to mirror it in-house and if OI is replaced by another system, it is possible to convert OIMs to plain *Java* libraries and reuse them in other developments.

With the examples in chapter 6, OIMs are available that provide the base functionalities for any neutron experiment and, as with the calibration example, a new feature is directly usable so that the setup time of experiments is significantly reduced and the quality and the reproducibility improved.

6.2.2. The Outcome for Experimenters and Observers

Experimenters will benefit in a similar way to the research facilities from the summary of instrument systems in chapter 2 because it helps selecting the software that is capable of performing a particular research experiment in the preferred way. If Open Inspire has been selected as experiment software, there are two ways for the user to prepare himself for the experiment, dependent on the privileges the administrator gives to the experimenter. The first way is to use a preconfigured experiment setup. The configurator in this case has already setup the whole experiment assembly and restricts the user being only able to start / stop the experiment, configure it using the controls provided by a preselected UI and collect the measurement data. The second way gives the user the flexibility to create a customized OI *Assembly*. For this purpose it is possible to browse the OI store, pick up desired OIMs and assemble them for a particular experiment assembly or simply modify an existing OI *Assembly*, which might come preconfigured by the instrument configurator or may be downloaded from the store. All OIMs hide their complexity behind simple interfaces and can be configured and interlinked through an XML-based *Assembly* file. Open Inspire is hence the only instrument system that allows the creation of highly customized experiment setups without programming knowledge. Only *LabView* has similar capabilities of allowing to graphically wire and configure instruments but *LabVIEW* is not a free *Open Source* product and does not come with a cloud-based component store. Beyond that, *LabVIEW* is not supported on all platforms in the same way[3].

Open Inspire in contrast can be used in the same way on *Windows*, *UNIX*, *Linux* and *Mac OS* and is freely installable on any computer. Experimenters can hence perform dry-runs on their own equipment and simply switch between a simulated and real hardware-based experiment to identify issues in advance. All OIMs and OILs stated in an *Assembly* are automatically downloaded so that a user does not need to take care if the software is identically setup on its own device. If a scan is running, the network service allows the monitoring of the experiment remotely from any device[4] at any location that has network access to the OI *Application Server*.

[3]Many extensions for example only run on *Windows* and are only partially supported on other platforms.
[4]A device can for instance be a computer, a smartphone or a tablet.

6.2.3. The Outcome for Administrators and Configurators

Administrators and *Configurators* especially benefit from Open Inspire's maintainability and security features.

Maintainability

Open Inspire is easy and quick to install due to its platform independency and thus removes the need to buy new hardware since it is runnable on existing instrument computers. To install OIMs and OILs, no additional work its necessary since they are automatically downloaded and installed from the central OI *Store*. The collaboration features allow the publishing of *Assemblies* as blueprints and the sharing of them with other *Configurators*. In most situations, only minor modifications such as the replacement of the *Hardware Proxy* OIMs are necessary to create copies of other instrument setups.

If Open Inspire is integrated into existing environments it is possible to migrate instruments step-by-step, to smoothly upgrade partial functionality and to operate new and legacy functionality in parallel. A legacy system such as *Caress* can hence remain in operation while a subset of functionality is carried out by OI thereby making use of exiting *Caress* hardware. Complete migrations in one step are thus unnecessary, which saves money, allows the use of functionality that cannot be migrated to OI and prevents the need for shutdowns to perform upgrades.

If functionality needs to be updated, OI automatically performs OIM and OIL updates in the background by automatically resolving all dependencies, downloading the correct version and installing them without any intervention and without the need to shutdown the system. If a new version of an OIM is requested by an *Assembly*, it can be installed separately beside older versions so that other *Assemblies* can still link to the version they are designed for.

Security and Stability

To prevent hardware damage or the unprivileged access to file resources, users can be restricted using the role management of OI's *Authentication Service* (page 140). Users can, for example, be restricted in their right to access the instrument from external networks, be

limited to access specific hardware or disallowed to edit *Assemblies*. The administrator can for instance restrict the user to the usage of a fixed set of preconfigured *Assemblies* without allowing them to create or modify them. Users need to login with a username and a password and obtain access to the functionality, which has been approved by the administrator. To prevent the read-out of passwords, they are stored in encrypted form.

To restrict the network access, OI intentionally uses only a limited number of fixed network ports that can be filtered, forwarded or easily tunnels through any firewall to prevent denial of service attacks or intrusion. If no internet access is available inside a restricted instrument network, it is possible to create a locally accessible mirror of the OI store, which also allows filtering particular unwanted modules and save bandwidth.

Due to the central bug reporting, issue tracking tools and continuous integration tools, it is not necessary to provide a local development infrastructure. Handling bugs and issues centrally allows the collaborative increase of code quality by reporting problems or submitting patches. If a bug caused invalid measurement results it is possible to scan *Assemblies* if a faulty OIM has been used and in particular situations to subsequently correct systematic errors in existing measurements files. This is only possible since a measurement can be associated with an *Assembly* that exactly reproduces the setup at the time of the measurement. Finding issues can be eased with help of the log files that are written during the execution of Open Inspire.

6.2.4. The Outcome for Developers and Software Engineers and Testers

Open Inspires (OIs) has been designed to be quickly and easily extensible without loosing flexibility. A developer can checkout OI from its *Mercurial Repository* using `hg clone http://hg.openinspire.org oi` and build it with the command `ant build`. OI automatically detects the OS and *architecture*, downloads platform specific dependencies and builds the system. If the developer wants to use an Integrated Development Environment, it is furthermore possible to let OI create all necessary project files to open it with the *NetBeans* IDE. With help of the preconfigured toolchain and IDE integration, it is possible to quickly start with the development and to make use of the advanced development features of *NetBeans*.

It is possible to setup the complete environment in less then 30 minutes, which matches requirement [☒ 2.3.4].

If an OIM is to be developed, a developer can clone an OIM template[5], select a device from the *Science Domain* and let *NetBeans* create all method bodies that are to be implemented with custom functionality. If an OIM outside the *Science Domain* shall be developed, it is only necessary to tag the methods that should act as interface to the outside with the *Annotations* `@OIPort` and `@OIProperty`. The experiment afterwards sees the OIM as a black box with several configuration properties and ports that allow the interlinking of OIMs to an *Assembly*. Setting the name, description, icon and configuring of the OIM's dependencies can be done using a properties file. No further proprietary interface definitions need to be created and no mandatory interfaces or communication functionality needs to be implemented, which allows a quick understanding of the concepts and enables an immediate start on the actual programming. The setup of a new OIM should hence not exceed 30 minutes and thereby fulfills requirement [☒ 2.3.5]. Understanding the relevant anatomy of an OIM should not take more than 15 minutes.

If an OIM or OIL is published over the OI *Store*, every developer can use the functionality and add it as dependency. Especially for OILs, this has advantages, since libraries can be shared between modules, only need to be downloaded once and can be updated globally so that every OIM that depends on an OIL automatically profits from bug fixes and enhancements. With the central issue tracker *JIRA* and code review tool *Crucible*, it is possible to collaboratively fix bugs, review code and suggest features and enhancements directly inside the code.

6.3. Outlook

While the design of Open Inspire is considered complete and requiring no further modifications, the implementation is a prototype and can thus be enhanced in several aspects. In the context of the thesis, it was necessary to concentrate on main aspects, so that several topics are only briefly touched on, to the extent to make OI work. This concerns in particular the UI implementations, a revision and extension of OI's *Domains* (page 123), an integration

[5]The name of the OIM *Template* is `org-openinspire-oim-templates-module`

of the simulation implementation (page 236) and the support for additional scripting and programming-languages.

6.3.1. OI Infrastructure Enhancements

The design and implementation of the OI *Application Server* is considered complete. Although the server is fast enough to allow an uninterrupted operation, there is still potential to speed up the container startup. With help of caching mechanisms, preloading, concurrent tasks as well as the usage of advanced libraries that improve the reflection and thus the injection process, it is still considered possible to speedup the container, so that it starts up to 40 times faster.

Concerning the OIMs and OIL it is meaningful to update all meta descriptions and to add a description of the module features and usage. Currently all descriptions are in english so that a translation could be added in multiple languages using a translation file. Some OILs come with a license file so that a mechanism needs to be added that allows users to accept it and comply with the license.

The prototype exclusively uses OIMs, which are written in *Java*. The design however also allows the use of any other of the scripting languages supported by the JSR 223 such as *Jython*, *Groovy*, *Ruby* or *Java Script* since they all compile to standard *Java* classes. To support these scripting languages, examples need to be provided showing how to create an OIM. To improve the development process, it is hence meaningful to provide a toolchain to build OIMs in other languages and to extend the IDE integration with support for the new type of OIMs. In an additional step it is even possible to run OIMs in completely different languages such as *C / C++*. Here it is however necessary to provide means that allow bridging between the interfaces of the custom language and *Java*.

Since custom languages cannot implement the *Java* interfaces of the *Domain Science* or other *Domains* it is necessary to generate interfaces for other languages or corresponding function bodies. Here it would be possible to switch to XML files that describe the interfaces of a *Domain* and use them to create the interfaces for the specific languages. *Java* interfaces how-

ever have the advantage that their information can be easily converted to other languages so that it is reasonable to keep them as standard interface for all conversions in other languages.

With this paradigm, OI would be the only system that is even language independent, allowing users to select their preferred language and enabling access and opening the system to the integration of existing functionality written in other languages.

Services and Configuration

The OI server uses a number of global services (see page 139), which are pluggable but only registrable directly on the code base. This makes it more complicated for users to, for example, replace the network model. Since these services are essential for the operation of the container, they cannot be simply dropped since they mandatorily need to be loaded at the server startup. Here it would be possible to move the functionality to OIMs and introduce a construct that allows the automatic loading of these OIMs independently from the *Container Lifecycle*, prior to when the server starts up.

THE INDIVIDUAL SERVICES COULD BE ENHANCED BY SEVERAL NEW FEATURES:

To improve the usage of the *Authentication Service*, a better interface could be developed that is accessible from OIMs to query the users privileges. A new example template could thus help to understand how to deal with user roles and privileges.

Improvements for the *Container Service* would be a more verbose feedback and control interface, which for example provides the client with descriptions and graphical symbols for the individual steps, which can be displayed when an *Assembly* is started, or more methods to interact with the container.

For the network service currently only one example is provided that uses an XML-based protocol for the data transfer. The advantage is that it is usable with all programming languages. The drawback is its overhead since it is text-based and hence not very efficient. Here it would be possible to use a binary protocol such as provided by *ZeroC Ice* [Hen04]. This middleware solution is based on a fast binary protocol and provides functionality to

generate interfaces for all common programming languages. In contrast to CORBA, it is more firewall friendly, because it can be simply restricted to a selected set of network ports.

The third global service is the *Logging Service.* This service writes log files dependent on the set log level. To access the log files from the client, the *FileSystemService* is used. This service could however be improved by adding the means to remotely query log files and apply dynamic filters to them.

The configuration is currently only accessible by editing the configuration files on the servers filesystem. To allow clients to configure the OI *Server* remotely, an interface could be added, which would allow access to the configuration by the client and provide an easy to use user interface, depending on the users privileges.

6.3.2. Library and Media Platform

To provide a central access to documentation and resources, the Open Inspire Library and Media Platform (OI-LAMP) has been introduced in section 4.5. While the infrastructure at `http://www.openinspire.org` is already in productive use, it still lacks content. This will change after the thesis is submitted, because parts of it will be published as PDF and HTML-based web documentation.

Furthermore it is planned to publish screencasts that show how to install OI on different platforms, how to configure it and how to develop OIMs. The current introductory videos (OI 2009) will be removed and replaced by videos that reflect the final state of OI. To quickly and easily make oneself familiar with OI, a preconfigured virtual machine will be published, which comes with multiple examples *Assemblies* and illustrations of the usage of OI in different scenarios.

The *Download* section allows the obtaining of prebuilt OI installers for multiple platforms. Currently customized installers are available for *Windows*, *Gentoo/Sabayon Linux* and platform independent installation packages for *Mac* OSX and all other *Linux* distributions. It is planned to extend the list with a native OSX installation package and packages for the common *Linux* distributions in form of `deb`-packages (*Debian*, *Ubuntu*, *Mint*) and `rpm` (*Red Hat*, *Centos*, *SUSE*). The automatic creation of installation packages, with the help of the

continuous build system, guarantees that all packages will always come in the latest version. Since OI is free *Open Source* software, it is possible to publish OI using distribution specific package databases.

As a new category, it is planned to publish the list of software systems from section 2.4 as a centrally maintained directory. The size of the thesis did not allow the evaluation of all systems, so that this directory would be a place to add some uncommon systems and extend it step-by-step with new developments.

6.3.3. Build System

The OI *Build System* is able to generate all files to open Open Inspire directly as a *NetBeans* IDE project. OI is though not bound to *NetBeans*, and it is an objective to provide additional generators that allow the creation of files for other IDEs such as *Eclipse* or *IDEA*.

All of these IDEs allow the performing of software tests and the profiling of applications to provide feedback about performance problems and memory leaks. The build system currently does not support *Unit Tests* and *Profiling* so that adding these features by means of two new build targets will be the next step in the enhancement of the *Build System*.

6.3.4. Example Applications

The thesis introduces a selected number of sample applications to prove the usability of the OI design. All of these applications are minimal examples and so there is still room for enhancements.

User Interfaces

The focus of the thesis was to develop the OI *Server* so that the UI's introduced in section 5.1 are only very limited examples that illustrate how a UI for Open Inspire could look like and how they could be integrated. For the overall usability of OI they are however essential so that the development of a full featured UI is an important target.

For a new development it is in the meantime more meaningful to use *Android*[6] as the target platform so that it is runnable on a tablet. A touch screen based UI simplifies the interaction with the instrument because it is more intuitive to manipulate a setting using a touchable widget. Moreover it is no longer necessary to control the experiment from a fixed place but will be possible to access the instrument from any location. A user can hence have a component such as an axis directly in his field of vision while being able to manipulate the device without the need to go to a fixed terminal. Further, monitoring the instrument when in transit is more convenient when only a smartphone or tablet need to be entrained.

A UI that provides access to the OI *Store* has been developed in *Java FX* since it allows execution as a standalone application or when embedded into the OI website. Oracle unfortunately decided to stop the development of *Java FX Script* so that the development of the OI *Store Browser* has also been canceled. A new graphical fronted to the OI Store is in development in form of a web application for the OI *Website*, as an *Android App* and as a *Java Desktop Client.*

NeXus Integration

When the *NeXus* integration for OIs was developed, it was necessary to wrap the original *C*-based *NeXus* API due to the lack of a means that allows to write HDF5 files in pure*Java.* In the meantime the *Java HDF5* writer *Nujan* [nuj11] has emerged, which allows the handling of HDF5 files without the need for a native library integration. Implementing the *NeXus* support entirely in *Java* allows the creation of a *NeXus* API with an *object oriented* interface, random access to the *NeXus* tree, the possibility to modify existing data and makes the native *NeXus* OILs obsolete. The implementation effort is however very high since all *NeXus* functionality need to be reimplemented.

Scan Implementations

The scan examples are as well as the UI implementations a minimal proof of concept. Here it is especially necessary to revise the *Sequencer Script* to make it more flexible and to provide UIs for the control and monitoring of the different scan engines.

[6]Other platforms are also conceivable but Android restricts developers least of all and supports Apps in Java.

Hardware Control with Caress

The *Caress* implementation is prepared to support any of the more than 100 different *Caress* devices. To have access to all these devices from an *Assembly* it is however necessary to provide additional *Device* OIMs beside the example OIMs for *Positioners*, *Counters* and *Detectors* to enable the access to devices such as *temperature controllers* or *binary switches.*

If OI is to be used at instruments that already use *Caress*, *instr*-files are available that contain the complete hardware configuration of the devices. Providing a tool, which is able to convert *Caress instr*-files to *OI Assemblies* is relatively easy to realize and saves the time required to configure all devices inside a newly created *Assembly* manually.

Simulation with McStas

Section 5.5.1 introduced two different methods for how *McStas* could be supported by Open Inspire. The example implementation has been realized by creating an interface to the *McStas Simulation Engine* and by using the native *McStas* configuration file for the simulation setup. Another solution is the modeling of the complete *McStas* experiment inside an OI *Assembly.* It is a future target to support such a modeling of *McStas* setups entirely by OI *Assemblies.*

Instrument Alignment

The instrument *E3* comes with a secondary slit, which is located directly in front of the detector and used to mask all neutrons around a selected area. This axes of this slit are however only manually adjustable. Adding *Positioners* to these axis would allow to add an additional step for the calibration of the secondary slit.

6.3.5. Introduction of new Domains

A fundamental paradigm of the OI *Design* is, that component interfaces are not interwoven with a framework but completely decoupled from the system. This is the essence underlying the flexibility because it is easy to use OI for completely different use-cases without needing to touch the framework.

This characteristic is especially important considering the community based standards for device interfaces that may be found in future. In this situation it would be possible to simply create a new *Domain* OIM, which defines the standard interfaces and to publish it using the OI store. Changes to the OI server are unnecessary. Even devices that build against the existing *Domain Science* (see 4.3.3) do not need to be changed. Here it would be possible to create an *Adapter* OIM that converts between the two different interface standards.

Open Inspire is decoupled from a specific use-case which makes it easy to define domains for new scenarios such as *Industrial Automation*, *Smart Home* or different scientific applications.

Figure 6.1.: *Open Inspire Smart Home Client*

A domain for *Home Automation* tasks can for instance contain *On Off Switches*, *Dimmer Switches*, *Window Contacts*, *Temperatur Control Devices*, *PID Controllers*, *Rain Sensors* etc.

Figure 6.1 shows the concept of a client that monitors several devices within a flat and allows the central switching of lights, control of radiators or even reacts to phone calls.

APPENDIX

A

SOFTWARE REQUIREMENTS SPECIFICATION

A.1. Look and Feel Requirements

The work of this thesis only includes the development of a minimal reference User Interface that exclusively covers the most important functions. The UI shall be easily exchangeable so that the following requirements both apply to the reference UI and every later independent UI developments.

A.1.1. Appearance and Style Requirements

☒ Requirement 1.1.1.

The UI shall be consistent within the whole application and resources.

A unified UI with a clear line allows the user to intuitively understand dialogs and controls by transferring the knowledge between already known and new dialogs.

✔ The whole application / webresources uses the same style with unified colors and fonts and only a small selection of recurring controls .

☒ Requirement 1.1.2.

The UI shall be consistent on all instruments, regardless of the operating system.

Users switch between different instruments and do not want to relearn the operation and interpretation of multiple different user interfaces.

✔ All instruments have the same Look and Feel and UI structures.

☒ Requirement 1.1.3.

The UI shall be clear and intuitively usable / understandable.

Users want to be able to use the UI without reading a manual.

✔ The UI shall be clearly structured, use common controls, come with a subtle style that only highlights the most important functionality and hides unnecessary details

☒ Requirement 1.1.4.

The system should be switchable between a consistent UI that is unique on all Operating Systems or a UI that matches the Operating System specific look and feel.

OS's have different LnFs so that a consistent LnF can mean a unique LnF between the developed application and other applications running on the OS or the same LnF of the application on all OS although it then differs from the OS's standard LnF.

✔ A user shall be able to switch between a system and a cross platform Look and Feel.

A.2. Usability Requirements

A.2.1. Ease of Use and Clarity Requirements

☒ Requirement 2.1.1.

The user shall not be overwhelmed with irrelevant information.

Users want to quickly understand dialogs and get the relevant information without searching for it.

✔ The software shall hide information that is not necessary for the current task. For instance use stepwise wizards instead of overloaded dialogs.

☒ Requirement 2.1.2.

The system should provide direct access to relevant information and detailed access on demand.

Normally only a subset of the overall functionality of the software is used so that complex or detailed information, which would empede the ease of use is hidden unless required.

✔ The user should only see the relevant information and decide if additional special functions should be made available.

☒ Requirement 2.1.3.

The system should give feedback about the validity and state of entered data.

Validating data when it is entered allows immediate error notifications, which saves time and avoids interrupting the workflow.

✔ The system should mark fields when they are invalid or show immediate information about the state of a control.

☒ Requirement 2.1.4.

The system should be easily configurable without programming and scripting skills.

Users that configure the software are not automatically developers.

✔ A user should be able to understand and change the system configuration without programming skills or an extensive system understanding.

A.2.2. Personalization and Internationalization Requirements

☒ Requirement 2.2.1.

The system shall come with preconfigured user profiles.

Specific settings and constraints shall be applicable dependent on the current user.

✔ The system provides user profiles that store individual user presets.

☒ Requirement 2.2.2.

The system shall come with preconfigured group profiles.

Groups are allowed to apply settings for a role instead of individual users, which lowers the administration effort, allows the constraining of user privileges and the hiding of secondary features.

✔ The system allows the application of role-based presets that apply to all users in the same group.

☒ Requirement 2.2.3.

The system shall allow users to personalize their user profile.

Users want to personalize their account settings, (such as language) and have easy access to customized data such as the history of recently opened files.

✔ A user is able to save individual settings to a persistent user profile and override approved group settings.

☒ Requirement 2.2.4.

The system shall allow the application of role based personalization.

Each group of users (see page 59 ff.) has specific demands on the software's functionality and presets.

✔ The system provides mechanisms that allow for role-specific settings concerning the user interface, accessible functionality or specific settings that override the presets.

☒ Requirement 2.2.5.

The system's default language shall be english.

English is spoken by almost every scientist or developer and the main language for publications.

✔ All dialogs, controls, descriptions text and documentation of the software and related developments are in english.

☒ Requirement 2.2.6.

The system should be easily internationalizable.

Some clients want a country specific localization and locale text encoding.

✔ Translations of the software can be added as localization files without changing the programming code and UTF-8 is used for the international text encoding.

A.2.3. Learning Requirements

☒ Requirement 2.3.1.

Experimenters shall be able to learn how to load, start and stop a prebuilt experiment in less then 15 minutes.

The time available for experimenters at an instrument is expensive so that performing the instrument main tasks need to be easy and quick to understand and perform.

✔ An experimenter understands in less then 15 minutes how to perform the base operations required to control an experiment.

☒ Requirement 2.3.2.

Configurators shall be able to learn how to setup a simple experiment in less than 30 minutes.

A main focus is the simplification of custom experiment setups so that it is important that an experiment can be intuitively setup in short time

✔ Configurators can learn how to setup a simple experiment in less than 30 minutes using a tutorial or screencast.

☒ Requirement 2.3.3.

Administrators shall be able to learn how to install the system in less than 30 minutes.

If the time for a standard system setup takes more than 30 minutes, the administrator is frustrated and the acceptance drops.

✔ An easy installer automates as much as possible and guides the administrator in less than 30 minutes through the setup of a basic system.

☒ Requirement 2.3.4.

Developers shall be able to learn how to create a simple extension module in less than 45 minutes

A developer must benefit from a framework in order to continue to use it. When the system is too complex, they will use other solutions or develop without a framework.

✔ Documentation shall be available that enables an understanding of how to develop an extension in less then 45 minutes.

☒ Requirement 2.3.5.

Developers should be able to learn how to use the development environment in less than 1h.

Developers are important to advance the system so that they should not flinch from using the system due to complex and time consuming setups of the development environment.

✔ Developers can configure their environment and start to develop for the system in less than 1h.

☒ Requirement 2.3.6.

The time needed to learn the API and system design shall be small against the time used for the development of modules.

The development becomes unproductive when the usage of the API and system takes far longer than the development of new functionality.

✔ The API and system design is mainly self-explanatory and easy to understand and supports the developers workflow.

☒ Requirement 2.3.7.

The system shall provide an interactive help that makes the program usable without prior training.

Software that comes with an interactive help saves time and improves the users motivation because it is often not necessary to read external documentation.

✔ The system comes with context sensitive help and descriptions directly where data is entered or information is queried.

☒ Requirement 2.3.8.

The system shall come with documentation that allow learning the system features step-by-step.

Step-by-step tutorials direct users through the main features of a system without the requirement to read a comprehensive manual.

✔ Tutorials are provided that allow to learn the primary system features in a guided tour.

A.2.4. Understandability and Politeness Requirements

☒ Requirement 2.4.1.

Users shall be able to instantly understand what the software will do for him.

The time necessary to read manuals can be cut down when a user is intuitively able to understand what a switch or function will do for him.

✔ The understanding of a feature directly emerges from the context or is explained beside the control or indicator.

☒ Requirement 2.4.2.

Users shall be able to understand the main system functions without training.

User often try to figure out functionality without having first read any instructions. Leading the user through the program by an intuitive operation increases the motivation to use the product.

✔ The system's main functionality is easily discoverable by trial and error.

☒ Requirement 2.4.3.

Informational and warning messages shall be clearly understandable by observers that are new to the system.

If a system error occurs but only staff such as a security agency is on location, messages must be clear enough to understand if it is a harmful misoperation and how it can be solved.

✔ Messages are clear and non-cryptic and contain a description that can be understand by unexperienced users.

☒ Requirement 2.4.4.

Wizards and dialogs should be used to limit the information, lead the user through the functionality and guide them to the important information.

If a description is not clear or too long it constrains the workflow and interrupts a user by requiring them to read external manuals.

✔ Descriptions are short and clear so that they are directly understandable. Additional information can be displayed on demand or by a link refering to detailed descriptions.

☒ Requirement 2.4.5.

The system shall not expect users to know and learn functions that are unrelated to their business problems.

Different groups of users have a different background and knowledge so that it makes sense that a user only needs to know the information that is really required to solve the problem and is not overwhelmed by nice but not necessary additional features that relate to other user groups.

✔ Dialogs and configuration files contain only the information, and require only the input that is absolutely necessary to solve the problem.

☒ Requirement 2.4.6.

The system shall hide implementation details and should not require users to learn the terms and details relating to the system internals.

Details of how a function is implemented internally are not relevant for an operation and should be hidden as otherwise they will detract from important features and usability will be impaired.

✔ The system comes with a facade that hides all implementation details and which is automated, as much as possible, in order to reduce the complexity.

☒ Requirement 2.4.7.

Advanced users should be able to enable additional complex functionality on demand.

The system hides all complex functionality but appropriate 'expert' users may be allowed access to additional features.

✔ Users can enable advanced features by activating an additional layer of expert functionality.

☒ **Requirement 2.4.8.**

The user shall be intuitively guided by naturally understandable symbols and texts.

Naturally understandable symbols help to explain functionality at one one glance, without imposing the need to read texts, and allowing complex features to be made more clear.

✔ Symbols and graphics complement descriptions to advance the quick understanding of system functionality.

A.3. Performance Requirements

A.3.1. Speed and Latency Requirements

☒ **Requirement 3.1.1.**

The System shall provide soft real-time capabilities in the range of seconds.

Hard real-time processing e.g. for the generation of motor control signals is provided by hardware servers or the hardware's firmware. The system to be developed shall be able to send and collect data at predetermined time intervals and perform soft realtime-tasks in the range of seconds. Time critical tasks are synchronized between hardware servers and not the Supervisory Control and Data Acquisition (SCADA) system.

✔ The system is able to trigger hardware or collect data within a range of less than 10 seconds.

☒ Requirement 3.1.2.

The System latencies shall be predictable so that the collection of one set of data can be scheduled to be finished before new data arrives.

The correct result at the wrong time is in real-time systems a wrong result. This requires that the processing time of tasks need to be predictable. A detector, for example, needs to be ready to detect before a new positioning is started.

✔ Data need to be reliable processible in predetermined time intervals (soft real-time).

☒ Requirement 3.1.3.

The System shall come with event mechanisms that allow the collection of data when it is first created in order to reduce delays.

Polling the hardware for available data creates unnecessary traffic when the trigger interval is small and reduces the responsiveness when it is too large.

✔ The system will be notified when a property changes or new data is available.

☒ Requirement 3.1.4.

Automated measurements shall be significantly faster than manual measurements.

The automation of functionality shall be a speed enhancement. When manual measurements are faster, it would be without advantage to run an automated experiment.

✔ The same functionality, such as the driving of an experiment, must be faster when it is automated than done in a manual way.

☒ Requirement 3.1.5.

The UI shall run on current computer hardware without noticeable latencies.

Latencies restrict the workflow and should only result from waiting times for important processes and never from the user interface being overloaded with dialogs or renderings that are not fast enough for current computers or mobile devices.

✔ The user interface is snappy and matches the speed of current computers or mobile devices.

☒ Requirement 3.1.6.

The system shall not loose responsiveness when large amounts of data are transferred or complex tasks are executed.

If the user interface is too slow to render information provided by system tasks it can become unresponsive and in the worst case block due to a growing data queue that cannot be processed fast enough.

✔ The system only renders important information and drops unimportant data packages when the rendering task is not able to draw the data fast enough.

☒ Requirement 3.1.7.

The system shall high internal throughput and low latencies, equivalent to monolithic applications, although all components are loosely coupled.

Loosely coupled components need to interchange data through defined interfaces. When these interfaces are a bottleneck (e.g. because of slow serialization and deserialization mechanisms) it is not possible to use the system for applications requiring fast or reactive processes.

✔ The component interfaces do not limit the throughput and guarantee small latencies.

A.3.2. Safety-Critical Requirements

☒ Requirement 3.2.1.

The system shall come with an emergency shutdown mechanism that stops all devices and tasks and brings the system to a safe state.

Misoperation of the system can be harmful so that a user must be able to immediately stop the system and all mechanical components to prevent damage and injuries.

✔ An emergency shutdown is available in all situations and stops all running devices and tasks with a minimum latency and high priority.

☒ Requirement 3.2.2.

The system shall be usable and work reliably in safety critical environments.

The system will run in security critical environments where failures can result in expensive delays, damages, or injuries.

✔ The system is designed to work reliably and comes with a crash protection, fallback functionality and redundant error prevention mechanisms.

☒ Requirement 3.2.3.

The system shall prevent damages caused by incorrect operation.

If user input is not sufficiently checked for its validity it can cause harm and damage during automatic operation.

✔ The system verifies all user input and directly prompts for corrections when invalid or when implausible data is entered.

☒ Requirement 3.2.4.

The system shall be fault-tolerant so that errors only affect restricted areas and do to spread across other parts of the software.

If subsystems are not cleanly separated from each other, errors can propagate between subsystems and effect the entire application.

✔ All subsystems are loosely coupled and mechanisms catch errors before propagation to other subsystems can occur.

☒ Requirement 3.2.5.

The system shall be able to perform unattended scans without causing any personal or property damage.

A main advantage of the system shall be the saving of time gained by the ability to run unattended scans, which requires a high degree of stability since users will not be in place to resolve problems.

✔ The system is designed to work unattended and comes with mechanisms that bring the system into a secure state, even when severe errors occur.

☒ Requirement 3.2.6.

The system must take care that users cannot destroy devices by improper handling.

Many damages have their origin in manual misoperation and can be avoided when the system comes with security mechanisms that detect and inhibit such mishandling.

✔ The system detects misuse and warns or blocks the user when harmful commands are applied.

☒ **Requirement 3.2.7.**

The system shall allow access to critical functionality to be restricted to selected users and groups.

Some functionality is often only reserved for advanced users and so access to this functionality should be restrictable.

✔ The system provides an authentication and authorization system which allows functionality to be restricted depending on the current user and group.

A.3.3. Precision or Accuracy Requirements

☒ **Requirement 3.3.1.**

The system shall be capable to perform precise scientific measurements.

The systems transfers data from and to other systems within the environment that this needs to be processed without information loss. To avoid precision-loss, all internal calculation shall be handled with a sufficient number of decimal places.

✔ The standard precision of internally used floating-point values is 32 bit and can be 64 when necessary.

☒ **Requirement 3.3.2.**

The UI shall display floating-point values in a legible way.

A user normally has no interest to read the complete number of decimal places but want to see a rounded and short visualization of the necessary information.

✔ While a floating-point value is internally processed with full precision, its visualization shall be rounded and rendered in a short form to keep the UI clearly readable.

☒ Requirement 3.3.3.

The UI shall allow high precision values to be viewed on demand.

Sometimes it is necessary to check a visualized value with a higher precision although it's standard visualized is in rounded form.

✔ Mouse-over effects allow high precision values to be displayed, when required.

A.3.4. Reliability and Availability Requirements

☒ Requirement 3.4.1.

The system shall work reliably, without crashing, over long periods of time.

When an instrument system is started, it will normally run for several months around the clock without restart. If the system crashes the work of several days, weeks or months may be waisted, leading to a loss of time and money.

✔ The core system is reliable 24 hours of the day, 365 days of the year, without restart.

☒ Requirement 3.4.2.

The state before an error occurred shall be recoverable.

When an error occurred it saves time and money when the measurement can be continued from the latest stable state.

✔ The system shall detect that an error occurred and continue the measurement when the system is recovered.

☒ Requirement 3.4.3.

The system should automatically restart if small errors occur.

If a measurement is running and only an insignificant error occurs the problem shall be bypassed to avoid the whole experiment failing.

✔ The measurement continues when a disfunction is exiguous.

☒ Requirement 3.4.4.

Migrating to the new system shall be smooth and cause minimum downtime.

Measurement time at the instrument is valuable so that downtimes need to be kept small.

✔ The migration can be done in parallel without obstructing the regular operation of the legacy system.

☒ Requirement 3.4.5.

Updates shall be able to be installed with minimal time effort in order to reduce downtime.

Installing updates interrupts the system operation. Updates will never be installed during an experiment run but should be installable without requiring the system to be restarted.

✔ Updates can be automatically installed while the system is running on the condition that no experiment is in progress.

A.3.5. Robustness and Fault-Tolerance Requirements

☒ Requirement 3.5.1.

Errors shall be limited to the subsystem where they occur.

Errors often affect not only the region where they appear but also compromise the functionality of other software subsystems or the complete software. This lack of decoupling can result in damage of affected subsystems.

✔ The system is separated into decoupled components and subsystems so that errors are contained to the region in which they occured.

☒ Requirement 3.5.2.

The system shall differentiate between errors and continue a measurement even when minor problems occur.

Measurements are often unattended and take several hours. When a minor error occurs and the system completely shuts down, the complete measurement time is lost so that a measurement should continue in case of minor problems and let the user decide subsequently if the result is still usable.

✔ The system continues a measurement in case of minor faults that cause no tangible or personal harm and logs / informs the user about the incident.

☒ Requirement 3.5.3.

The system shall continue with all unaffected parts if network errors occur and try to recover the connection.

Networks can have spontaneous latency or bandwidth problems that are mostly temporary.

✔ The system shall continue with all calculations that are not affected from the network problem, inform the user and wait until the network is recovered.

☒ Requirement 3.5.4.

The system shall immediately recognize or correct erroneous inputs.

When user input is checked at the moment it is entered instead of the moment it is used, errors can be avoided earlier.

✔ User input is directly checked when it is entered, so that the user can be prompted for corrections before the input is used in a processing task.

A.3.6. Capacity Requirements

☒ Requirement 3.6.1.

The system shall be able to process and store very large quantities of data.

Modern detection systems produce an increasing amount of data that needs to be stored and transferred in time. Instrument *E2* at the HZB for instance creates more than $4GB$ data per measurement. This is too large to address in one step with a 32 bit system.

✔ The system shall be able to process and store data with more than $4GB$ using 64 bit addressing or streams.

☒ Requirement 3.6.2.

The system shall allow the simultaneous handling of multiple user network client connections.

An important difference to classic systems that only have one integrated user interface is the ability to connect and get instrument data from multiple clients at the same time.

✔ A controlling user and multiple observers can connect to the system at the same time and send commands or query data.

☒ Requirement 3.6.3.

The system shall be able to process tasks of multiple components or subsystems at the same time.

It is possible to improve the measurement speed, when the next step is already calculated or simulated while the current measurement step is still in progress.

✔ The system allows to concurrently execute and synchronize multiple tasks in loosely coupled components.

A.3.7. Scalability and Extensibility Requirements

☒ Requirement 3.7.1.

The system shall be easily and quickly extensible.

A scientific environment often requires frequent changes to setups and adjustments that cannot be predicted in advance. The software needs to be flexible enough to implement new features in a short time.

✔ The system comes with a component model that allows to quickly add new features without touching the core system.

☒ Requirement 3.7.2.

The system shall be highly scalable.

System limitations such as the fixed size of datatypes, array lengths, buffer sizes or the maximal addressable space may make sense at the time a feature is developed but they limit scalability when the system requires new functionality.

✔ The size of datatypes, arrays, buffers as well as time constraints such as minimum latencies or maximal bandwidths are not artificially limited.

☒ Requirement 3.7.3.

The extension system and communication between components should not be a bottleneck in the transfer of data.

A component-based system needs to interchange data between components. Depending on the communication mechanisms used this can be slow and can consume much computational power. The serialization and deserialization of data is for example slower than shared-memory based transfers.

✔ The data interchange between components of the extension system is of a similar speed to the data-transfer in monolithically applications.

☒ Requirement 3.7.4.

The update mechanism and infrastructure shall be designed to scale with an increasing number of users.

Users download components and updates from a central repository whose availability and scalability is important to guarantee a fast, secure and reliable operation of the system. If the system does not scale with the number of users, it is a bottleneck that will impair productivity.

✔ The update system can be easily scaled by adding new mirrors and by balancing the load between them.

A.3.8. Longevity Requirements

☒ Requirement 3.8.1.

The system shall be expected to operate for a minimum of 10 years.

Instrument systems often have lifetimes of more than 10 or 20 years due to the fact that the purchase of expensive hardware and the time for the setup of the complex experiment environment must be worthwhile. The instrument software is thus developed for the whole instrument lifecycle and even updates are installed rarely. An important rule is "Never touch a running system" to prevent unwanted side effects and avoid additional training.

✔ The system is still usable after 10 years and can be easily adopted to new problems or future requirements.

☒ Requirement 3.8.2.

The system should still be compatible with legacy plugins and configurations after upgrades or updates are performed.

Updates often affect interfaces or connection mechanisms and make old system components unusable. This wasts valuable development time and prevents updating just because of compatibility problems with legacy components.

✔ Legacy components or other components may still be used together with new functionality when updates are performed.

☒ Requirement 3.8.3.

The system shall be kept up-to-date with the help of regular incremental updates.

Small updates are easier to test and are more easily rolled-back when problems occur than major changes. If updates occur automatically it is possible to provide fast patches and avoid the delay of waiting for aggregated updates.

✔ Fine grained updates are made globally available and automatically installed in preference to the installation of new versions of the overall system.

☒ Requirement 3.8.4.

The system documentation shall be kept up-to-date.

When the system development goes on, the corresponding documentation always needs to be synchronized with the current system functionality and APIs.

✔ The provided documentation reflects the current development state and is kept up-to-date.

☒ Requirement 3.8.5.

The system shall allow for the recovery of an older system state if a failure occurs.

Misuse can destroy the system integrity. If no backup is available, recovering classical systems is complex or they need to be reinstalled.

✔ Components, dependencies and configurations can be recovered, deleted or invalid components will be detected and automatically reinstalled.

A.4. Operational and Environmental Requirements

A.4.1. Expected Physical Environment

☒ Requirement 4.1.1.

The system shall be operable in safety-critical areas with limited physical access.

The software runs in radioactive environments and may control devices that can cause personal injury. Due to the radiation it is not possible to access all parts of the instruments during experiments.

✔ The system continues to operate reliable and is remotely maintainable when a direct access is not possible due to security restrictions.

A.4.2. Requirements for Interfacing with Adjacent Systems

☒ Requirement 4.2.1.

The system must be able to control the hardware of the instruments, introduced in section 2.3 on page 24.

The instruments *E3* and *E7* at the HZB have been selected for the reference implementation and are available as platform for development tasks.

✔ The system is able to control and query data from all devices at the instruments *E3* and *E7* at BER-II and *StressSpec* at FRM-II

☒ Requirement 4.2.2.

The system must be able to communicate with *Caress* Hardware Servers.

All devices at the research diffractometers *E3* and *E7* (2.3.2) and nearly all devices at the HZB use *Caress* to interface the hardware.

✔ The system allows to interface *Caress* Hardware Servers, control devices and query data.

☒ Requirement 4.2.3.

The system must be able to create *NeXus* compatible output.

The standard format for neutron diffraction experiment results is *NeXus*.

✔ It is possible to create valid *NeXus* files with all measurement data collected during a diffraction scan.

A.4.3. Productization Requirements

☒ Requirement 4.3.1.

The system shall come with installers for all major operating systems

Operating system specific installers provide a fast way to cleanly install and deinstall software while meeting all operating system specific particularities.

✔ The system comes with installers for *Windows*, *Linux* and *Mac OSX*.

☒ Requirement 4.3.2.

The system shall be published in the form of platform independent packages.

Platform independent packages allow the system to be used on architectures and OSs where no explicit installer is available. Every operating system has a preferred archive standard so that archives should be published in multiple formats to increase the chance to unpack it under all OSs

✔ Platform independent packages with all data required to run the system are published as *zip*, *tgz* and *tbz* archives.

☒ Requirement 4.3.3.

The core system shall be installable as standalone without the need for other dependent software.

Collecting, downloading, installing and keeping dependent software up-to-date complicates the installation process, increases the maintenance effort and evokes version conflicts and compatibility problems.

✔ The system runs on all major platforms and automatically resolves all dependencies for the specific OS and architecture.

☒ Requirement 4.3.4.

The main package shall be small enough to be downloaded in short time, even with slow internet connections.

Instrument systems

✔ The installation package is not bigger than 10MB.

☒ Requirement 4.3.5.

The installation shall only contain all absolutely necessary multi-platform data and reload additional platform specific data during the installation.

If the main installation package only contains the functionality that is absolutely necessary to run the system and additional components can be downloaded on demand, it is possible to reduce download times and lower the required bandwidth.

✔ The installation package only contains the main functionality while additional functionality is downloaded on demand.

A.4.4. Release Requirements

☒ Requirement 4.4.1.

The system shall come with a completely automated build and release system.

Building, packaging and delivering the software manually, takes much time so that this task is often performed rarely so that users need to wait for aggregated updates. Using a continuous delivery always keeps the user up-to-date with the latest version even in case of small fixes or minor improvements.

✔ A continuous delivery system is triggered when code changes and automatically builds, tests, packages and uploads the system.

☒ Requirement 4.4.2.

The latest installation packages shall always be ready to be freely downloaded through a public website.

Users do not want to search for the latest updates but prefer one fixed central place where they know to download always the latest version. Third-party systems shall be able to have one fixed link that always points to the latest version.

✔ The latest installable system binaries are always downloadable from a fixed place on the project webpage.

☒ Requirement 4.4.3.

Major versions shall be published at regular time intervals.

In addition to the automatically delivered versions, publishing major versions allows to integrate larger numbers of fixes and new features that are ready to be publishes as a whole. These builds are generally more stable and consistent since they underly more comprehensive quality checks and manual tests.

✔ A major version with aggregated fixes and improvements shall be manually tested and publishes in intervals of 6 months.

☒ Requirement 4.4.4.

The latest sources shall be publicly available.

The system is an Open Source development and everyone shall be able to work with the latest version, inspect the quality or make changes.

✔ The latest sources are downloadable in the form of archives from the project webpage or directly browseable and downloadable from the VCS.

A.5. Maintainability and Support Requirements

A.5.1. Maintenance Requirements

☒ Requirement 5.1.1.

The application of updates shall not take longer than 10 minutes on average.

The core system should be small, lightweight and without any ballast that makes updates take longer than absolutely necessary. Small updates consume less time and are less fault-prone.

✔ The download, extraction and installation time with a internet connection faster than 10 MBit takes not longer than 10 minutes.

☒ Requirement 5.1.2.

The installation of components / plugins shall not take longer than the installation of a core update on a monolithic system.

When a system is built from multiple loosely coupled components rather than in form of a monolithic system, installing only the individual updates that are necessary for the specific system decreases download time and required storage space.

✔ The average installation / update time for components is small against the time for updates of monolithic systems.

☒ Requirement 5.1.3.

The creation and configuration of component assemblies / instrument setups shall be fast and intuitively performable.

The main advantage of the developed system shall be the flexible, easy and fast creation and configuration of system setups that can be created and modified by users without extensive training.

✔ Users without programming skills are able to setup an experiment setup / component assembly using prebuilt components.

☒ Requirement 5.1.4.

The installation of updates and components shall be performable from users with little software experience.

When the installation of updates can be performed automatically, it is not necessary to have specially trained administrators that keep the system up-to-date.

✔ Necessary updates will be initiated downloaded, configured and performed automatically without user interaction when they are required.

A.5.2. Supportability Requirements

☒ Requirement 5.2.1.

Software support shall be provided by the community using web-based community services.

The project is intended to be a free *Open Source* project so that cost-free services can only be offered by people that provide support with no charge. Web-based collaboration services shall help to aggregate all community services at a central place.

✔ Active help is provided by community members using a web-based community portal.

☒ Requirement 5.2.2.

Web-based forums shall be provided as main collaboration service.

Forums allow community members to ask questions and obtain answers from the community without charge. The advantage compared to mailing lists is the persistence of the responses, so that historic browse threads can be searched for existing answers to questions. In addition this format provides the ability to format code, or insert meta-data.

✔ A central forum is available through a web portal that is grouped into topics for beginners or developers. Questions will be answered between community members and regularly checked by advanced users and developers.

☒ Requirement 5.2.3.

Frequently Asked Questions (FAQs) shall be collected and answered centrally.

Many questions are asked again and again so that it makes sense to collect these questions and publish them on a website with the respective answers. This keeps the forum free from redundant answers for the same question and allows the publishing of a correct and comprehensive answer.

✔ A section with Frequently Asked Questions aggregates all common questions and their answers at a central place and is shipped with the installation packet.

☒ Requirement 5.2.4.

It shall be possible to get the exact state and configuration of the system for debugging purposes.

It is often hard to find an error when the exact configuration cannot be reproduced. This is why it is important to have a possibility to obtain information about the exact setups that has a problem.

✔ It is possible to output the system configuration, installed components and current state when a problem occurs.

A.5.3. Adaptability Requirements

☒ Requirement 5.3.1.

The system should be directly usable without changing the hardware of existing instrument environments.

If the instrument software of an existing environment shall be migrated to the new system it should be unnecessary to change the hardware or software interfaces to save money and time.

✔ The system is directly able to communicate with already existing instrument hardware.

☒ Requirement 5.3.2.

The system shall be platform independent.

Running the software on existing instrument computers eases the migration and prevent investments in new hardware. Making the software available on as many systems as possible opens it to a wide range of users.

✔ The system runs on common OSs *Windows*, *Linux*, *Mac OS* and *UNIX* systems such as *BSD* or *Solaris* both on 32 and 64 bit systems.

☒ Requirement 5.3.3.

The system shall be easily internationalizable.

While the software's standard language shall be english, it should be prepared to be easily adaptable to other languages without the need for source code changes. This enables new languages to be added quickly when customer specific regulations prescribe other languages than english.

✔ The system comes with internationalization support that allows the adding of new languages using translation files which include country-specific notations such as the different delimiter symbols.

☒ Requirement 5.3.4.

The system shall be operable directly at the instrument, from the office and while in transit between the two.

For the setup of an experiment it is necessary to have direct access to the instrument and to run a user interface directly at the experiment. During the experiment run it is normally not necessary to interfere with the experiment so that a monitoring should be possible from the office that provides a more secure and comfortable working environment. When an experiment is running over night is makes sense to be able to monitor the experiment from home, or while traveling, to get immediate feedback in case of failures.

✔ The network mechanisms allow secure connection, via firewalls, to the system from anywhere and comes with clients for the common computer OSs and mobile devices.

☒ Requirement 5.3.5.

The system shall allow for simple interfacing to third party systems, independent from their network implementation and protocols.

The system is intended to be a middleware that bridges different instrument system and hardware solutions so that systems with different network and communication implementations need to be addressable.

✔ It is possible to communicate with different third-party instrument systems and hardware that all build on different network mechanisms at the same time.

☒ Requirement 5.3.6.

The system shall reuse already existing free software if possible.

Reinventing the wheel and reimplementing already existing free third-party software is ineffective and costs development and maintenance time.

✔ When free-third-party software already provides required functionality, it will be incorporated or interfaced.

☒ Requirement 5.3.7.

The system shall be adaptable to be compatible with future third-party instrument systems and hardware.

Systems mostly come with one or a limited set of communication protocols that allow hardware to be addressed. Adding new protocols is either not possible, or requires comprehensive changes which makes it hard to adopt the system to future environment.

✔ The system is not restricted to a limited set of protocols and communication mechanisms but open to quickly adding communication mechanisms so that future instrument hardware may be addressed.

☒ Requirement 5.3.8.

The system shall be quickly adaptable to future standards without the need for core changes.

Fixed communication interfaces, that are interweaved with the system core, are difficult to change in response to new standards. For this reason changes often make the system incompatible with software that builds on the legacy interface definitions.

✔ The interfaces definitions required for the communication with third-party systems are decoupled from the system core.

☒ Requirement 5.3.9.

The system shall provide components that can be reused in third-party systems.

The usability and acceptance of a component standard increases when they are also usable in other independent software systems.

✔ The components are decoupled from the system core so that the can be reused in third-party systems.

A.6. Security Requirements

A.6.1. Access Requirements

☒ Requirement 6.1.1.

The system shall be protectable against unprivileged access.

Driving devices or executing specific functionality from unprivileged users can cause harm or destroy components. If functionality can be restricted to selected users, access to critical functionality can be constrained.

✔ The system comes with user and group based permission profiles that allow to restrict functionality to specific users or user groups.

☒ Requirement 6.1.2.

The system shall forbid all users except for administrators to change the system configuration.

The system configuration includes sensible data such as user configurations or access privileges and should be restrictible to administrators with elevated privileges.

✔ Only users that are part of a preinstalled administrator group have write access to the system configuration.

☒ Requirement 6.1.3.

The network-based access to the system shall only be allowed after a positive user authentication.

If the network-based access to the system is not restricted, anyone can query or change data remotely, which results in a major security problem.

✔ The remote access to the system is only possible after a successful user authentication.

☒ Requirement 6.1.4.

The system shall restrict observers to the querying of public information.

Observers can be public users and should only have readable access to general information so that they cannot read protected data or manipulate the system.

✔ Observers are restricted to have read-only access to public data.

A.6.2. Integrity Requirements

☒ Requirement 6.2.1.

The system shall catch and verify erroneous user input directly when it is entered.

When a user submits faulty information for a process that is running later, the problem is only discovered when the process is running. This results in a crash or stop until correct data will be submitted and hinders an autonomous operation.

✔ User input is verified directly when it is submitted to prevent starting misconfigured operation.

☒ Requirement 6.2.2.

The system shall allow easy recovery of a former state of the system from backups.

If a misconfiguration or data-loss prevents the operation of the system it is often necessary to reinstall the instrument software or spend much time to bring the system back to the last working state.

✔ If an error occurs, the system allow the automatic recovery of the system to the last working state.

☒ Requirement 6.2.3.

The system shall allow the reconstruction of the exact instrument setup at the time of an experiment when an experiment configuration file is loaded.

Recovering the exact software setup of a previous experiment, even when years old, is often impossible due to the ongoing development of the system and the resulting incompatibilities.

✔ Loading a setup / instrument configuration causes the system to automatically recover the exact state of the system when the configuration was created.

☒ Requirement 6.2.4.

The system shall allow systematical setup or software errors to be discovered and corrected when they become known.

When a series of measurements has been performed and it is announced that the setup or software was faulty its hard to correct old measurements because the data does not contain the exact software setup applicable at the time of the measurement.

✔ The description file for an instrument setup contains the exact software setup of the experiment and is added to the measurement results.

A.6.3. Privacy Requirements

☒ Requirement 6.3.1.

The system shall store and transfer safety critical user data such as passwords in encrypted form.

If the transferred data is not encrypted, the system is vulnerable to man-in-the-middle attacks where attackers are able to get confidential data by sniffing the network traffic.

✔ All data that is transferred over the network between the system and other system is encrypted.

☒ Requirement 6.3.2.

The system shall be configurable to meet the security requirements of the operator.

For many operators it is important to meet specific requirements concerning the security configuration of software so that these properties should be easily configurable without changing the software itself.

✔ The software comes with configurable security settings such as the encryption of user data or transport streams.

A.6.4. Audit Requirements

☒ Requirement 6.4.1.

The system shall come with a protocol / logging mechanism.

A logbook helps debugging the system by finding and analyzing malfunction, discovering break-in attempts or creating usage statistics. An example is the logging of logins or logouts to discover unprivileged access attempts or the logging during measurements to comprehend the complete measurement process.

✔ The system comes with a logger for system messages, warnings and errors.

☒ Requirement 6.4.2.

The logging entries shall contain all information necessary to comprehend, filter and analyze the logged issue.

Incomplete descriptions of issues make it hard to analyze a problem so that the time, logging-level (INFO, WARNING, ERROR, SEVERE), a description and the location of the issue (component / class / method) should be added to a log entry.

✔ Each log message contains the time, log-level, type of message, the logging message itself and the program part where the logged issue occurred.

A.6.5. Immunity Requirements

☒ Requirement 6.5.1.

The system shall be easily shieldeable against attacks from the outside.

When software comes with non-transparent communication mechanisms and opens many dynamical ports or uses uncommon protocols, it is hard to filter malicious packages by firewalls.

✔ The network protocols for the main system can be easily filtered, they are tunnable and the number of required ports is kept small.

A.7. Legal Requirements

A.7.1. Compliance Requirements

☒ Requirement 7.1.1.

The system must be published as Open Source Product.

A cooperation with the International Atomic Energy Agency (IAEA) makes it necessary to make all sources available. When all sources are available everyone can assure himself that the software comes with no backdoors or security flaws and is able to change or add functionality.

✔ All source codes are freely accessible and published under a Open Source Initiative (OSI) compliant license.

☒ Requirement 7.1.2.

🗊	The system must be free of dependencies to commercial third-party products.
☆	When commercial third-party code is included into the system, it cannot be guaranteed that the software can be freely published without violating any third part rights and it is not possible to publish the complete source.
✔	The system does not violate any patent and license rights and exclusively includes or links to free third-party libraries or modules.

☒ Requirement 7.1.3.

🗊	The system must configurable to comply with the guidelines of the operating institute or company.
☆	Clients often have guidelines that forbid specific functionality that for example breaks local security rules.
✔	All system functionality can be flexibly configured to the operators guidelines or be disabled.

A.7.2. Standards Requirements

☒ Requirement 7.2.1.

🗊	The system shall use the *NeXus* format to store neutron diffraction data.
☆	The *NeXUs* format is the standard format used to store and interchange neutron diffraction measurement data.
✔	It is possible to save measurement data in the *NeXus* format.

APPENDIX

B

FILES

B.1. OI Assemblies

B.1.1. E3-McStas.xml

```
<oi-assembly  ... >
<!-- The McStas Proxy -->
    <oim id="proxy" type="org-openinspire-oim-mcstas-proxy"
       version="1.3.0_alpha">
        <property name="simulationFolder" value="/path/to/E3sim" />
        <property name="instrumentFileName" value="E3.instr" />
        <property name="executableFileName" value="E3.out" />
    </oim>

<!-- The Positioners -->
    <oim id="TTHS" type="org-openinspire-oim-mcstas-positioner"
       version="1.3.0_alpha">
        <property name="value" value="90.0" />
        <property name="minimum" value="-120.0" />
        <property name="maximum" value="120.0" />
        <port name="mcStasProxy" ref="proxy" />
    </oim>

    <oim id="OMGS" type="org-openinspire-oim-mcstas-positioner"
       version="1.3.0_alpha">
        <property name="minimum" value="-360.0" />
        <property name="maximum" value="360.0" />
        <port name="mcStasProxy" ref="proxy" />
    </oim>

    <oim id="XT" type="org-openinspire-oim-mcstas-positioner"
       version="1.3.0_alpha">
        <property name="minimum" value="-200.0" />
```

```
25          <property name="maximum" value="200.0" />
26          <port name="mcStasProxy" ref="proxy" />
27      </oim>
28
29      <oim id="YT" type="org-openinspire-oim-mcstas-positioner"
            version="1.3.0_alpha">
30          <property name="minimum" value="-200.0" />
31          <property name="maximum" value="200.0" />
32          <port name="mcStasProxy" ref="proxy" />
33      </oim>
34
35      <oim id="ZT" type="org-openinspire-oim-mcstas-positioner"
            version="1.3.0_alpha">
36          <property name="minimum" value="-200.0" />
37          <property name="maximum" value="200.0" />
38          <port name="mcStasProxy" ref="proxy" />
39      </oim>
40
41      <oim id="CHIS" type="org-openinspire-oim-mcstas-positioner"
            version="1.3.0_alpha">
42          <property name="minimum" value="-200.0" />
43          <property name="maximum" value="200.0" />
44          <port name="mcStasProxy" ref="proxy" />
45      </oim>
46
47      <oim id="PHIS" type="org-openinspire-oim-mcstas-positioner"
            version="1.3.0_alpha">
48          <property name="minimum" value="-200.0" />
49          <property name="maximum" value="200.0" />
50          <port name="mcStasProxy" ref="proxy" />
51      </oim>
52
53  <!-- The Counters -->
54      <oim id="neutronCounter"
            type="org-openinspire-oim-mcstas-counter"
            version="1.3.0_alpha">
55          <port name="mcStasProxy" ref="proxy" />
56      </oim>
57
58      <oim id="progressCounter"
            type="org-openinspire-oim-mcstas-counter"
            version="1.3.0_alpha">
59          <port name="mcStasProxy" ref="proxy" />
60      </oim>
61
62  <!-- The Detector -->
63      <oim id="detector" type="org-openinspire-oim-mcstas-detector"
            version="1.3.0_alpha">
64          <property name="nX" value="256"/>
65          <property name="nY" value="256"/>
66          <port name="mcStasProxy" ref="proxy" />
67      </oim>
68
69  <!-- The Scan Engine -->
```

```
70      <oim id="genericScan" type="org-openinspire-oim-scans-generic"
            version="1.3.0_alpha">
71          <port name="positioners" ref="TTHS" />
72          <port name="positioners" ref="OMGS" />
73          <port name="positioners" ref="XT" />
74          <port name="positioners" ref="YT" />
75          <port name="positioners" ref="ZT" />
76          <port name="positioners" ref="CHIS" />
77          <port name="positioners" ref="PHIS" />
78
79          <port name="counters" ref="neutronCounter"/>
80          <port name="counters" ref="progressCounter"/>
81
82          <port name="detectors" ref="detector" />
83
84          <port name="activePositioner" ref="TTHS"/>
85          <port name="activeCounter" ref="neutronCounter"/>
86          <port name="activeDetector" ref="detector"/>
87
88          <property name="fileNamePrefix" value="scan"/>
89          <property name="scanName" value="One␣Axis␣Diffraction␣Scan"/>
90          <property name="activePositionerStartValue" value="85.0"/>
91          <property name="activePositionerEndValue" value="95.0"/>
92          <property name="countsPerStep" value="5000000"/>
93          <property name="steps" value="6"/>
94      </oim>
95
96  <!-- The Summation Module -->
97      <oim id="debyeSum"
            type="org-openinspire-oim-math-sum-debyesherrer"
            version="1.3.0_alpha">
98          <property name="wavelength" value="1.374"/>
99          <property name="distance" value="0.7"/>
100         <port name="detector" ref="detector" />
101         <port name="positioner" ref="TTHS" />
102     </oim>
103
104 <!-- The visualization -->
105     <oim id="widgetTestcenter"
            type="org-openinspire-oim-debug-widgettestcenter"
            version="1.3.0_alpha">
106         <port name="components" ref="detector" />
107         <port name="components" ref="TTHS" />
108         <port name="components" ref="OMGS" />
109
110         <port name="components" ref="neutronCounter" />
111         <port name="components" ref="progressCounter" />
112         <port name="components" ref="genericScan" />
113         <port name="components" ref="debyeSum" />
114     </oim>
115
116 </oi-assembly>
```

B.2. Scan Configuration Files

B.2.1. scanSequence.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<sequence>
    <init>
        <nexus id="nx" filename="scan.hd5" type="hdf5"/>
        <nexus cmd="addgroup" name="entry1" type="NXentry"/>
        <nexus cmd="addgroup" name="instrument" type="NXinstrument"
           parent="entry1"/>
    </init>
    <steps>
        <step>
            <oim id="xT" call="setpoint">0</oim>
            <oim id="yT" call="setpoint">0</oim>
            <oim id="zT" call="setpoint">0</oim>
            <oim id="omgs" call="setpoint">0</oim>
            <block-until state="IDLE">xT,yT,zT,omega</block-until>
            <oim id="detector" call="clearMatrix"/>
            <oim id="counter" call="resetCounts"/>
            <oim id="counter" call="setSetpoint">0</oim>
            <oim id="detector" call="startDetecting"/>
            <oim id="counter" call="startCounting"/>
            <block-until state="IDLE">counter</block-until>
            <oim name="counter" call="stopCounting"/>
            <oim name="counter" call="stopDetecting"/>
            <nexus cmd="snapshot"
               parent="instrument">xT,yT,zT,omgs,detector</nexus>
        </step>
        <step>
            <oim id="omgs" call="setpoint">20</oim>
            <block-until state="IDLE">omega</block-until>
            <oim id="detector" call="clearMatrix"/>
            <oim id="counter" call="resetCounts"/>
            <oim id="detector" call="startDetecting"/>
            <oim id="counter" call="startCounting"/>
            <block-until state="IDLE">counter</block-until>
            <oim name="counter" call="stopCounting"/>
            <oim name="counter" call="stopDetecting"/>
            <nexus cmd="snapshot"
               parent="instrument">omgs,detector</nexus>
        </step>
        <step>
            <oim id="omgs" call="setpoint">40</oim>
            <block-until state="IDLE">omega</block-until>
            <oim id="detector" call="clearMatrix"/>
            <oim id="counter" call="resetCounts"/>
            <oim id="detector" call="startDetecting"/>
            <oim id="counter" call="startCounting"/>
            <block-until state="IDLE">counter</block-until>
            <oim name="counter" call="stopCounting"/>
            <oim name="counter" call="stopDetecting"/>
            <nexus cmd="snapshot"
               parent="instrument">omgs,detector</nexus>
```

```
48          </step>
49          <step>
50              <oim id="omgs" call="setpoint">60</oim>
51              <block-until state="IDLE">omega</block-until>
52              <oim id="detector" call="clearMatrix"/>
53              <oim id="counter" call="resetCounts"/>
54              <oim id="detector" call="startDetecting"/>
55              <oim id="counter" call="startCounting"/>
56              <block-until state="IDLE">counter</block-until>
57              <oim name="counter" call="stopCounting"/>
58              <oim name="counter" call="stopDetecting"/>
59              <nexus cmd="snapshot"
                   parent="instrument">omgs,detector</nexus>
60          </step>
61          <step>
62              <oim id="omgs" call="setpoint">80</oim>
63              <block-until state="IDLE">omega</block-until>
64              <oim id="detector" call="clearMatrix"/>
65              <oim id="counter" call="resetCounts"/>
66              <oim id="detector" call="startDetecting"/>
67              <oim id="counter" call="startCounting"/>
68              <block-until state="IDLE">counter</block-until>
69              <oim name="counter" call="stopCounting"/>
70              <oim name="counter" call="stopDetecting"/>
71              <nexus cmd="snapshot"
                   parent="instrument">omgs,detector</nexus>
72          </step>
73          <step>
74              <oim id="omgs" call="setpoint">100</oim>
75              <block-until state="IDLE">omega</block-until>
76              <oim id="detector" call="clearMatrix"/>
77              <oim id="counter" call="resetCounts"/>
78              <oim id="detector" call="startDetecting"/>
79              <oim id="counter" call="startCounting"/>
80              <block-until state="IDLE">counter</block-until>
81              <oim name="counter" call="stopCounting"/>
82              <oim name="counter" call="stopDetecting"/>
83              <nexus cmd="snapshot"
                   parent="instrument">omgs,detector</nexus>
84          </step>
85          <step>
86          <nexus id="nx" cmd="save" group="entry1" />
87          </step>
88      </steps>
89  </sequence>
```

B.3. CARESS Files

The following pages give an overview about configuration files used as information and design source for the Open Inspire CARESS support modules. All templates are captured from the setup of the instrument E7 but are similar on the instruments E3 and StressSpec.

B.3.1. hardware_modules_e7.dat

```
;+++++++++++++++++++++++++++++++++++++++++++++++++++++++++
;
;   E7 hardware components
;
;   K.H. Degenhardt
;   S. Flemming
;   14-jun-2005
;---------------------------------------------------------

;components controlled by server running on nvmee7
SLAVE1   1000

;JOERGER VSC counter
;name    kind     bus     addr        no
MON      116      11      0x1000000   1
TIM1     116      11      0x1000000   2
TIM2     116      11      0x1000000   3

#if ADET
;X18 histogram module
;name kind  bus    addr       low/high_pattern   x_chan   y_chan
ADET  121    11 0X8000000     0X400       128    128
NYS   0
YSD   0
#endif

#if LDET
; JOERGER counter
;name    kind     bus  addr          offset  chans
LDET     113      11   0xe6000000    0       5
YSD   0
;single detector selection
;name    kind  x_chan    y_chan    region
D1       109   1         0         0
#endif

#if SDET
;single detector
;name    kind    bus addr        no
SDET     116     11  0x1000000   4
#endif
#if ALL
MON2     116     11  0x1000000   4
#endif

;IO24 I/O module
;name kind  bus addr  input_mask
IOREG 120    11  0xffff6000  0x0

;motor control EKF 44520
;name    kind  bus  addr        no.   ratio   vmax  accel  enct  reso  dx_max  dir  pads  PID  umax  scale  P   I  D
      o_shoot
OMGS     114   11   0xf0f1c000  1     -4096   2000    200     2    24      50    0     0     1  3000   1   10  0  0  0
TTHS     114   11   0xf0f1c000  2      4096   2000    200     2    24      50   -1    11     1  3000   1   10  0  0
OMGM     114   11   0xf0f1d000  2      4096  10000   1000     2    24     100    1     0     1  3000   1   10  0  0

#if EULER |ALLMOTS | ALIGNM | ALL
PHIS     115   11   0xf0f1d000  4      3200   2000    200     1     0       0    0     0     1  5000   1   10  0  0  0
CHIS     115   11   0xf0f1d000  3     -6400   2000    200     1     0       0    0     0     1  5000   1   10  0  0  0
#endif

;standard translation modules
XT      114   11    0xf0f1c000  3      8192  10000   1000     2    24     100   -1     0     1  5000   1   10  0  0  0
YT      114   11    0xf0f1c000  4      8192  10000   1000     2    24     100   -1     0     1  5000   1   10  0  0  0
ZT      114   11    0xf0f1d000  1     16000  10000   1000     2    24     100   -1     0     1  5000   1   10  0  0  0

#if TRANS
XE      115   11    0xf0f1e000  1      3200  10000   1000     1     0       0    0     0     1  5000   1   10  0  0  0
YE      115   11    0xf0f1e000  2      3200  10000   1000     1     0       0    0     0     1  5000   1   10  0  0  0
ZE      115   11    0xf0f1e000  3     32000  10000   1000     1     0       0    0     0     1  5000   1   10  0  0  0
#endif

if ALIGNM | ALL
;PST
;PSR
;SST
#endif

;motor control multiplex ST180
;name kind  bus   term  unit#
MUX3  38 4  /t4   1
;MUX modules (attention: velo and accel in steps per s, CARESS converts these
;natural unit into the obscure ST180 unit)
;name kind  mux   lun   motor#   ratio velo  accel    cof     cli
M1_CHI  39     3       1       1       -400.0   200     100
M1_FOC  39     3       1       2       -400.0   200     100
M1_TR   39     3       1       3        400.0   200     100
M2_CHI  39     3       1       4       -400.0   200     100
M2_FOC  39     3       1       5       -400.0   200     100
M2_TR   39     3       1       6        400.0   200     100
M3_CHI  39     3       1       7       -400.0   200     100
```

```
M3_FOC  39     3       1       8       -400.0   200     100
M3_TR   39     3       1       9        400.0   200     100

NALE7  1000

; Keithley DMM199 multimeter
;name kind  bus   addr  gpibadr  ch#   fact. settl.   func. range
HE_CM   27     2       5       16      1      0.012   0       0      2

#if TEMP
;LakeShore temp controller via DIGI interface at ALPHA
;name kind  bus   lun   xxx   xxx   xxx
TEMP    25     2       5       12      14      15
#endif
```

B.3.2. params_e7.com

```
;+++++++++++++++++++++++++++++++++++++++++++++++
;
; param7_e7.com
;
; K.H. Degenhardt
; S. Flemming
; last update: 14-jun-2005
;
;-----------------------------------------

; define instrument type and mode
EXPTYPE E7 A2 ADET ALL TRANS TEMP MAGF
USN USER
TITLE TITLE
RELA TTHS=128,64,0.151181 NYS=128,64,0.151181 YSD=0

;define parameters for drive components
BLS OMGS=-200,200 TTHS=-200,200 PHIS=-200,200 CHIS=-200,200 OMGM=-200,200
BLS XT=-200,200 YT=-200,200 ZT=-200,200 XE=-200,200,YE=-200,200 ZE=-200,200
BLS PSR=-200,200 PST=-200,200 SST=-200,200 TEMP=0,100 MAGF=0,5
BLS TTHM=-200,200 TRANSM=-200,200 M1_TR=-600,600 M1_CHI=-600,600 M1_FOC=0,4096
BLS M2_TR=-600,600 M2_CHI=-600,600 M2_FOC=0,4096 M3_TR=-600,600 M3_CHI=-600 600
BLS M3_FOC=0,4096
TOL OMGS=0.01 TTHS=0.01 PHIS=0.01 CHIS=0.01 OMGM=0.01 XT=0.01 YT=0.01 ZT=0.01
TOL XE=0.01 YE=0.01 ZE=0.01 PSR=0.01 PST=0.01 SST=0.01 TEMP=0.01 MAGF=0.01
TOL TTHM=0.01 TRANSM=0.01 M1_TR=0.01 M1_CHI=0.01 M1_FOC=0.01 M2_TR=0.01
TOL M2_CHI=0.01 M2_FOC=0.01 M3_TR=0.01 M3_CHI=0.01 M3_FOC=0.01
SOF OMGS=0 TTHS=0 PHIS=0 CHIS=0 OMGM=0 XT=0 YT=0 ZT=0 XE=0 YE=0 ZE=0
SOF PSR=0 PST=0 SST=0
SOF TTHM=0 TRANSM=0 M1_TR=0 M1_CHI=0 M1_FOC=0 M2_TR=0 M2_CHI=0 M2_FOC=0
SOF M3_TR=0 M3_CHI=0 M3_FOC=0
TMO OMGS=60 TTHS=60 PHIS=60 CHIS=60 OMGM=60 XT=60 YT=60 ZT=60
TMO XE=60 YE=60 ZE=60 PSR=60 PST=60 SST=60 TEMP=60 MAGF=60
TMO TTHM=60 TRANSM=60 M1_TR=60 M1_CHI=60 M1_FOC=60 M2_TR=60 M2_CHI=60 M2_FOC=60
TMO M3_TR=60 M3_CHI=60 M3_FOC=60

; define sample parameter
SAM SAMPLE
WAV 1.7
SET YSD=50

WINDA W1=1 128 1 128

PRT ON
DISP ON
DISP LIV 10

EXTERN
```

```
//++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
//
// CORBA IDL file for CARESS "abstract device".
//
// omniidl  -bcxx -Wba absdev.idl
//
// file: absdev.idl
// author:  K.H. Degenhardt
// last update: 29-mar-2000
//
//
//--------------------------------------------------------------------

#ifndef __ABSDEV_IDL__
#define __ABSDEV_IDL__

// max. items in a data block
const long MAX_ITEMS = 4096;

// define module info
typedef sequence<any> module_info_seq_t;

// define block data info
```

```
typedef sequence<char> char_data_seq_t;
typedef sequence<short> short_data_seq_t;
typedef sequence<long> int_data_seq_t;
typedef sequence<float> float_data_seq_t;

interface absdev
{
  // init system
  long init_system_orb(in long kind_of_init_system, out long system_status);

  // release system
  long release_system_orb(in long kind_of_release_system);

  // init module
  long init_module_orb(in long kind_of_init_module, in long module_id, in string module_line, out long
       module_status);

  // read module (out replaced by inout)
  long read_module_orb(in long kind_of_read_module, in long module_id, inout module_info_seq_t module_info_seq);

  // drive module
  long drive_module_orb( in long kind_of_drive_module, in module_info_seq_t module_info_seq, in long event_number,
       inout long calculated_timeout, out long event_used, out long delay_status, out long module_status);

  // load module
  long load_module_orb(in long kind_of_load_module, in module_info_seq_t module_info_seq, in long event_number,
       out long event_used, out long module_status);

  // stop module
  long stop_module_orb(in long kind_of_stop_module, in long module_id, out long module_status);

  // stop all modules
  long stop_all_orb(in long kind_of_stop_all, out long stop_all_status);

  // start acquisition
  long start_acquisition_orb(in long kind_of_start_acquisition, in long run_no, in long mesr_count, out long
       acquisition_status);

  // stop acquisition
  long stop_acquisition_orb(in long kind_of_stop_acquisition, out long acquisition_status);

  // get parameters of block data
  long readblock_params_orb( in long kind_of_readblock_module, in long module_id, inout long start_channel, inout
       long end_channel, inout long channel_type);

  // read block data for char, short, long or float
  long char_readblock_module_orb(in long kind_of_readblock_module, in long module_id, in long start_channel, in
       long end_channel, out long module_status, inout char_data_seq_t data_seq);

  long short_readblock_module_orb(in long kind_of_readblock_module, in long module_id, in long start_channel, in
       long end_channel, out long module_status, inout short_data_seq_t data_seq);

  long int_readblock_module_orb(in long kind_of_readblock_module, in long module_id, in long start_channel, in
       long end_channel, out long module_status, inout int_data_seq_t data_seq);

  long float_readblock_module_orb(in long kind_of_readblock_module, in long module_id, in long start_channel, in
       long end_channel, out long module_status, inout float_data_seq_t data_seq);

  // load block data for char, short, long or float
  long char_loadblock_module_orb(in long kind_of_loadblock_module, in long module_id, in long start_item, in long
       end_item, in long item_type, out long module_status, in char_data_seq_t data_seq);

  long short_loadblock_module_orb(in long kind_of_loadblock_module, in long module_id, in long start_item, in long
       end_item, in long item_type, out long module_status, in short_data_seq_t data_seq);

  long int_loadblock_module_orb(in long kind_of_loadblock_module, in long module_id, in long start_item, in long
       end_item, in long item_type, out long module_status, in int_data_seq_t data_seq);

  long float_loadblock_module_orb(in long kind_of_loadblock_module, in long module_id, in long start_item, in long
       end_item, in long item_type, out long module_status, in float_data_seq_t data_seq);

  // read all modules (out replaced by inout)
  long read_allmodules_orb( in long kind_of_read_allmodules, inout module_info_seq_t module_info_seq);
};

#endif
```

B.4. McStas Instrument Files

B.4.1. E3.instr

```
/*
 * Instrument: E3 at HMI (Residual Stress Diffractometer)
 * written by: Mirko Boin
 * data collected by: Robert Wimpory
```

```
 * last change: 30.05.2008
 */

/* possibility to set motors and slit dimensions */
DEFINE Instrument E3( TTHS=90.0, OMGS=45.0, XT=0.0, YT=0.0, ZT=0.0,
   CHIS=0.0, PHIS=0.0 )

DECLARE
%{
/* initialize instrumental defaults here */
double monochromator_tth = 27.3064; // (111) Si reflection
double monochromator_sampleTable_distance = 2.350;
double tubeSlit_monochromator_distance = 0.85;
double tubeSlit_width = 0.11;
double tubeSlit_height = 0.11;

/* this usually belongs to the experiment (sample) */
double primarySlit_width = 0.02;
double primarySlit_height = 0.02;
double secondarySlit_width = 0.02;
double secondarySlit_height = 0.02;
double primarySlit_sampleTable_distance = 0.15;
double secondarySlit_sampleTable_distance = 0.15;
double sampleTable_detector_distance = 1.300;
%}

INITIALIZE
%{
%}

TRACE

/* status of the progress while simulating */
 COMPONENT origin = Progress_bar(
      profile = "progress.txt",
      //percent=1,
      flag_save=1,
      minutes=0.01
 ) AT (0,0,0) ABSOLUTE

/* source */
COMPONENT source = Source_div(
  width=0.01,
  height=0.01,
  hdiv=0.002,
  vdiv=0.002,
  Lambda0 = 1.48,
  dLambda = 0.01,
  gauss = 0
)  AT (0, 0, 0) ABSOLUTE

/* slit or collimator? or both? */
COMPONENT tubeSlit = Slit(
  width = tubeSlit_width,
  height = tubeSlit_height
```

```
59 ) AT (0, 0, 1) RELATIVE source ROTATED (0, 0, 0) RELATIVE source
60
61 /* monochromator (here: perfectly bent focussing) */
62 COMPONENT monochromator_center = Arm() AT (0, 0,
      tubeSlit_monochromator_distance) RELATIVE tubeSlit ROTATED (0, 0,
      0) RELATIVE tubeSlit
63   COMPONENT monochromator = Monochromator_flat(
64     width = 0.2,
65     height = 0.2,
66     mosaich = 30.0,
67     mosaicv = 30.0,
68     r0 = 0.9,
69     Q = 2.004
70   ) AT (0, 0, 0) RELATIVE monochromator_center ROTATED (0,
        -monochromator_tth/2.0, 0) RELATIVE monochromator_center
71
72 COMPONENT monochromator_out = Arm() AT (0, 0, 0) RELATIVE
      monochromator_center ROTATED (0, -monochromator_tth, 0) RELATIVE
      monochromator_center
73
74 /* primary slit */
75 COMPONENT primarySlit = Slit(
76   width = primarySlit_width,
77   height = primarySlit_height
78 ) AT (0, 0,
      monochromator_sampleTable_distance-primarySlit_sampleTable_distance)
      RELATIVE monochromator_out ROTATED (0, 0, 0) RELATIVE
      monochromator_out
79
80 /* sample table including eulerian cradle */
81 COMPONENT sampleTable_center = Arm() AT (0, 0,
      primarySlit_sampleTable_distance) RELATIVE primarySlit ROTATED
      (0, 0, 0) RELATIVE primarySlit
82   COMPONENT omgs_motor = Arm() AT (0, 0, 0) RELATIVE
        sampleTable_center ROTATED (0, OMGS, 0) RELATIVE
        sampleTable_center
83     /* X, Y, Z tables */
84     COMPONENT zt_motor = Arm() AT (0, ZT, 0) RELATIVE omgs_motor
          ROTATED (0, 0, 0) RELATIVE omgs_motor
85     COMPONENT xt_motor = Arm() AT (XT, 0, 0) RELATIVE zt_motor
          ROTATED (0, 0, 0) RELATIVE zt_motor
86     COMPONENT yt_motor = Arm() AT (0, 0, -YT) RELATIVE xt_motor
          ROTATED (0, 0, 0) RELATIVE xt_motor
87       /* Eulerian Cradle (chis, phis) */
88       COMPONENT chis_motor = Arm() AT (0, 0, 0) RELATIVE yt_motor
            ROTATED (0, 0, -CHIS) RELATIVE yt_motor
89       COMPONENT phis_motor = Arm() AT (0, 0, 0) RELATIVE chis_motor
            ROTATED (0, PHIS, 0) RELATIVE chis_motor
90         /* --- XE, YE, ZE motors could also be implemented here --- */
91         /* finally, the sample */
92         COMPONENT sample = Powder1(
93           d = 1.0465,
94           xwidth = 0.13,
95           yheight = 0.1,
96           zthick = 0.05,
```

```
97            d_phi = 10,
98            radius = 0
99          ) AT (0, 0, 0) RELATIVE phis_motor ROTATED (0, 0, 0) RELATIVE
              phis_motor
100 COMPONENT tths_motor = Arm() AT (0, 0, 0) RELATIVE sampleTable_center
      ROTATED (0, TTHS, 0) RELATIVE sampleTable_center
101
102 /* secondary slit */
103 COMPONENT secondarySlit = Slit(
104   width = secondarySlit_width,
105   height = secondarySlit_height
106 ) AT (0, 0, secondarySlit_sampleTable_distance) RELATIVE tths_motor
      ROTATED (0, 0, 0) RELATIVE tths_motor
107
108 /* 2D detector */
109 COMPONENT detector = PSD_monitor(
110   xwidth = 0.3,
111   yheight = 0.3,
112   nx = 256,
113   ny = 256,
114   filename = "detector.psd"
115 ) AT (0, 0, sampleTable_detector_distance) RELATIVE tths_motor
      ROTATED (0, 0, 0) RELATIVE tths_motor
116
117 END
```

BIBLIOGRAPHY

[AdRF04] L. Alianelli, M.S. del Rio, and R. Felici. A Monte Carlo algorithm for the simulation of Bragg scattering by imperfect crystals; and II. Application to mosaic copper. In *Proceedings of SPIE*, volume 5536, page 27, 2004. 44

[alb09] ALBA Synchrotron Light Facility [online]. 2009. URL: `http://www.cells.es/` [cited 2009-08-02]. 33

[Ale01] B. Alexander. Linux/UNIX-Shells. 2001. 181

[All03] O.S.G. Alliance. *OSGi Service Platform, Release 3.* IOS Press, Inc., 2003. 86

[ALN] P.O. Astrand, K. Lefmann, and K. Nielsen. The philosophy of McStas. 46

[ALN01] PO Astrand, K. Lefmann, and K. Nielsen. Monte Carlo simulations of neutron scattering instruments. *Hungarian Academy of Sciences Central Research Institute for Physics-Publications-KFKI*, pages 27–34, 2001. 44

[ALS+a] F. A. Akeroyd, Ashworth R. L., Campbell S.I., Johnston S. D., Moreton-Smith C. M., Sergeant R.G., and Sivia D. S. Open Genie - Reference Manual. URL: `http://www.isis.rl.ac.uk/GenieBinaries/TheReferenceManual-2.0.pdf` [cited 2009-08-22]. 42

[ALS+b] F. A. Akeroyd, Ashworth R. L., Campbell S.I., Johnston S. D., Moreton-Smith C. M., Sergeant R.G., and Sivia D. S. The Open Genie User Manual. URL: `http://www.isis.rl.ac.uk/GenieBinaries/TheUserManual-1.1.pdf` [cited 2009-08-22]. 42

[ALTK+04] P. Andersen, K. Lefmann, L. Theil Kuhn, PK Willendrup, and E. Farhi. Monte Carlo simulations as a part of the configuration for neutron instruments. *Physica B: Physics of Condensed Matter*, 350(1-3S):721–724, 2004. 44

[ank] ANKA - The synchrotron facility at the Forschungszentrum Karlsruhe (KIT) [online]. URL: `http://ankaweb.fzk.de/` [cited 2009-08-20]. 38

[anl] Argonne National Laboratory [online]. URL: `http://www.anl.gov/` [cited 2009-08-05]. 35

[ans] ANSTO - Australian Nuclear Science and Technology Organisation [online]. URL: `http://www.ansto.gov.au/` [cited 2009-08-06]. 36, 37

[apa11a] The Apache Ant Buildsystem Website [online]. 05 2011. URL: `http://ant.apache.org/` [cited 2011-05-18]. 155, 161

[apa11b] The Apache Maven Buildsystem Website [online]. 05 2011. URL: `http://maven.apache.org/` [cited 2011-05-18]. 155

[Arn05] S.L. Arnold. Open Source Technologies in Science Education: What's Your Geek IQ? In *21st International Conference on Interactive Information Processing Systems*, 2005. 89

[Bal01] H. Balzert. *UML kompakt: mit Checklisten.* Spektrum Akademischer Verlag GmbH, 2001. 14

[Bal05] H. Balzert. *Lehrbuch der Objektmodellierung: Analyse und Entwurf mit der UML 2. 2.* Spektrum Akademischer Verlag GmbH, 2005. 14

[BBvB+01] K. Beck, M. Beedle, A. van Bennekum, A. Cockburn, W. Cunningham, M. Fowler, J. Grenning, J. Highsmith, A. Hunt, R. Jeffries, et al. Manifesto for agile software development. 11:2004, 2001. URL: `http://www.serena.com/docs/agile/papers/manifesto-for-agile-software-development.pdf` [cited 2009-08-12]. 37

[Bin79] K. Binder. *Monte-Carlo Methods.* Wiley Online Library, 1979. 175

[BK04] H. Bässmann and J. Kreyss. Bildverarbeitung ad Oculos. 2004. 247

[BK06] A. Brule and O. Kirstein. Residual stress diffractometer KOWARI at the Australian research reactor OPAL: Status of the project. *Physica B: Physics of Condensed Matter*, 385:1040–1042, 2006. 37

[BLFM05] T. Berners-Lee, R. Fielding, and L. Masinter. RFC 3986: Uniform resource identifier (uri): Generic syntax. *The Internet Society*, 2005. 90

[Bol03] A. Bolour. Notes on the Eclipse Plug-In Architecture. *Bolour Computing, July*, 3, 2003. 86

[Bou02] T. Boudreau. *NetBeans: The Definitive Guide.* O'Reilly Media, 2002. 86

[BPHLB06] P.M. Bentley, C. Pappas, K. Habicht, and E. Lelièvre-Berna. Evolutionary programming for neutron instrument optimisation. *Physica B: Physics of Condensed Matter*, 385:1349–1351, 2006. 44

[BRJ06] G. Booch, J. Rumbaugh, and I. Jacobson. Das UML Benutzerhandbuch: Aktuell zur Version 2.0, 2006. 14

[Bun85] H.J. Bunge. Experimental techniques of texture analysis. *Clausthal, FRG, 25-29 Mar. 1985*, page 1986, 1985. 23

[caj11] The cajo project: Wiki: Home — java.net [online]. 05 2011. URL: `http://java.net/projects/cajo/pages/Home` [cited 2011-05-07]. 141

[CAK+99] J-M. Chaize, A.Götz, W-D. Klotz, J.Meyer, M.Perez, and E.Taurel. TANGO - An object oriented control system based on CORBA. In *ICALEPCS 1999*, ESRF, BP220, Grenoble, 38043, FRANCE, 1999. ESRF. URL: `http://www.tango-controls.org/Documents/papers/icalepcs1999.pdf/` [cited 2009-08-03]. 33

[cam] Introduction to CAMAC [online]. URL: `http://www-esd.fnal.gov/esd/catalog/intro/introcam.htm` [cited 2009-10-05]. 30

[CAMS02] SI Campbell, FA Akeroyd, and CM Moreton-Smith. Open GENIE-Analysis and Control. *Arxiv preprint cond-mat/0210442*, 2002. URL: http://arxiv.org/pdf/cond-mat/0210442 [cited 2009-08-22]. 42

[Car95] JA Carwardine. An introduction to plant monitoring through the EPICS control system. In *International conference on accelerator and large experimental physics control systems, Chicago, IL (United States), 30 Oct-3 Nov 1995*, 1995. URL: http://www-bd.fnal.gov/icalepcs/abstracts/PDF/wpo6.pdf [cited 2009-08-05]. 34

[CGK+01] J-M Chaize, A. Götz, W-D. Klotz, J. Meyer, M. Perez, E. Taurel, and P. Verdier. The ESRF TANGO Control System Status. In *ICALEPCS 2001*, ESRF, BP220, Grenoble, 38043, FRANCE, 2001. ESRF. URL: http://www.tango-controls.org/Documents/papers/icalepcs2001.pdf [cited 2009-08-03]. 33

[CJS+04] G. Chiozzi, B. Jeram, H. Sommer, A. Caproni, M. Plesko, M. Sekoranja, K. Zagar, D.W. Fugate, P. Di Marcantonio, and R. Cirami. The ALMA common software: a developer-friendly CORBA-based framework. In *Proceedings of SPIE*, volume 5496, pages 205–218, 2004. URL: http://www.eso.org/~gchiozzi/MyDocs/SPIE2004/SPIE2004-5496-23.pdf [cited 2009-08-27]. 38

[CKM01] J. Castro, M. Kolp, and J. Mylopoulos. A Requirements-driven Development Methodology. In *Advanced Information Systems Engineering*, pages 108–123. Springer, 2001. 56

[CMM+02] A. Chatterjee, D. Mikkelson, R. Mikkelson, J. Hammonds, and T. Worlton. Remote access and display of neutron data. *Applied Physics A: Materials Science & Processing*, 74:194–197, 2002. URL: http://www.springerlink.com/content/ghkgl2ydwdxpawyw/fulltext.pdf?page=1 [cited 2009-08-23]. 42

[Coc02] A. Cockburn. *Agile software development.* Addison-Wesley Boston, MA, 2002. URL: http://www.snip.gob.ni/xdc/Agile/AgileSoftwareDevelopment.pdf [cited 2009-08-12]. 37

[com] TechSmith | Camtasia Screen Recorder Software, Home [online]. URL: http://www.techsmith.com/camtasia/ [cited 2011-09-17]. 149

[cru] Webpage of the Code Review Tool - Crucible [online]. URL: http://www.atlassian.com/software/crucible/ [cited 2011-09-17]. 154

[CSH+02] T. Chatterji, R. Schneider, J. U. Hoffmann, D. Hohlwein, R. Suryanarayanan, G. Dhalenne, and A. Revcolevschi. Diffuse magnetic scattering above TC in quasi-two-dimensional La1.2Sr1.8Mn2O7. *Physical Review B (Condensed Matter and Materials Physics)*, 65(13):pp. 134440/1–6, 2002. 43

[css] Control System Studio (CSS) Website [online]. URL: www.cs-studio.org [cited 2009-08-17]. 39

[CT] M. Clausen and G. Tkacik. EPICS Office. *Proceedings of ICALEPS2005, Geneva.* URL: https://accelconf.web.cern.ch/accelconf/ica05/proceedings/pdf/05_010.pdf [cited 2009-08-11]. 37

[cvb12] Common Vision Blox [online]. 08 2012. URL: http://www.commonvisionblox.com/ [cited 2012-08-28]. 245, 247

[CWH+00a] A. Chatterjee, T. Worlton, J. Hammonds, C.K. Loong, D. Chen, and D. Mikkelson. Integrated spectrum analysis workbench (ISAW): A web-based neutron scattering data visualization tool. In *American Physical Society, Annual March Meeting, March 20-24, 2000 Minneapolis, MN, abstract# Y32. 012*, 2000. URL: http://www.osti.gov/energycitations/servlets/purl/751897-PyDvbs/webviewable/751897.pdf. 42

[CWH+00b] A. Chatterjee, T. Worlton, J. Hammonds, C.K. Loong, D. Chen, and D. Mikkelson. Integrated spectrum analysis workbench (ISAW): A web-based neutron scattering data visualization tool. In *American Physical Society, Annual March Meeting, March 20-24, 2000 Minneapolis, MN, abstract# Y32. 012*, 2000. 42

[CWH+01] A. Chatterjee, T. Worlton, J. Hammonds, C.K. Loong, D. Chen, D. Mikkelson, and R. Mikkelson. Neutron Scattering Data Access, Visualization and Analysis

Tool, ISAW. In *American Physical Society, Annual March Meeting, March 12-16, 2001 Washington State Convention Center Seattle, Washington Meeting ID: MAR01, abstract# V32. 014*, 2001. 42

[CZ+05] G. Chiozzi, K. Zagar, et al. The ALMA Common Software (ACS): status and developments. *Geneva, Switzerland: ICALEPCS*, 2005. URL: https://accelconf.web.cern.ch/accelconf/ica03/PAPERS/MP703.PDF [cited 2009-08-21]. 38

[dana] DANSE Software Release Site [online]. URL: http://danse.us/ [cited 2009-08-05]. 35

[danb] Main Page - DANSE [online]. URL: http://wiki.cacr.caltech.edu/danse/index.php/Main_Page [cited 2009-08-05]. 35

[dat] Dataflow Programming Basics in NI LabVIEW - National Instruments [online]. URL: http://www.ni.com/gettingstarted/labviewbasics/dataflow.htm [cited 2011-08-21]. 95

[dav] DAVE, Data Analysis and Visualization Environment [online]. URL: http://www.ncnr.nist.gov/dave/ [cited 2009-08-25]. 43

[DDE+04] JA Dann, MR Daymond, L. Edwards, JA James, and JR Santisteban. A comparison between ENGIN and ENGIN-X, a new diffractometer optimized for stress measurement. *Physica B: Physics of Condensed Matter*, 350(1-3S):511–514, 2004. 39, 202

[des09] Deutsches Elektronen-Synchrotron DESY [online]. 2009. URL: http://zms.desy.de/ [cited 2009-08-02]. 34, 35

[DHK+93] LR Dalesio, JO Hill, M. Kraimer, S. Lewis, D. Murray, S. Hunt, M. Claussen, W. Watson, and J. Dalesio. The experimental physics and industrial control system architecture: past, present, and future. In *Conference: ICALEPCS93: International conference on accelerators and large experimental physics control systems, Berlin (Germany), 18-22 Oct*

1993, 1993. URL: http://www.osti.gov/energycitations/servlets/purl/10193541-orZnty/native/10193541.pdf [cited 2009-08-05]. 34

[DKK91] L. R. Dalesio, M. R. Kraimer, and A. J. Kozubal. EPICS Architecture. 1991. URL: http://www.aps.anl.gov/epics/EpicsDocumentation/EpicsGeneral/EPICS_Architecture.pdf [cited 2009-08-05]. 34

[DKO+] J. Dovc, M. Kadunc, Y. Oku, M. Plesko, et al. NEW ABEANS FOR TINE JAVA CONTROL APPLICATIONS. URL: http://www.slac.stanford.edu/econf/C011127/THAP025.pdf [cited 2009-08-17]. 38

[DPSR09] K. H. Degenhardt, E. E. Pape, O. P. Sauer, and L. Rossa. *Caress Hardware Descriptions.* Helmholtz-Centre Berlin for Materials and Energy, April 2009. 209, 217, 228

[DS06] V. Desnica and M. Schreiner. A LabVIEW-controlled portable x-ray fluorescence spectrometer for the analysis of art objects. *X-Ray Spectrometry*, 35(5), 2006. 41

[DSHT99] L.L. Daemen, P.A. Seeger, R.P. Hjelm, and T.G. Thelliez. Monte Carlo tool for neutron optics and neutron scattering instrument design. In *Proceedings of SPIE*, volume 3771, page 80, 1999. 44

[DTT+08] N. J. Draper, R. J. Taylor, R. Tolchenov, M. J. Clarke, A. J. Markvardsen, S. Nagella, R. Fowler, F. A. Akeroyd, L. C. Chapon, P. G. Radaelli, and T. G. Perring. Mantid: Manipulation and Analysis Toolkit for ISIS Data. In *NOBUGS 2008 Conference*, 2008. 43

[ebu] Gentoo Linux Documentation – Ebuild HOWTO [online]. URL: http://www.gentoo.org/proj/en/devrel/handbook/handbook.xml?part=2&chap=1 [cited 2010–11-08]. 89

[ecl11] The Eclipse Project Webpage [online]. 05 2011. URL: http://www.eclipse.org/ [cited 2011-05-29]. 169

[ele09] Sincrotrone Trieste S. C. p. A. [online]. 2009. URL: http://www.elettra.trieste.it/ [cited 2009-08-02]. 34

[Eng97] R. Englander. *Developing Java Beans.* O'Reilly Media, 1997. 191

[ER02] T. Erler and M. Ricken. UML-Das bhv Taschenbuch. *Aufl. Bonn: Verlag moderne Industrie Buch AG & Co. KG*, 2002. 14

[ER05] T. Erler and M. Ricken. UML 2-gepackt, 2005. 14

[eso] ESO, the European Southern Observatory [online]. URL: http://www.eso.org [cited 2009-08-27]. 38

[esr09] ESRF - A Light for Science [online]. 2009. URL: http://www.esrf.eu/ [cited 2009-08-02]. 34

[FAA] Brent Fultz, Michael A. G. Aivazis, and Ian S. Anderson. Distributed Data Analysis for Neutron Scattering Experiments CNST. URL: http://wiki.cacr.caltech.edu/danse/images/0/01/DANSE_CNST_txt.pdf [cited 2009-08-05]. 35

[FHSL99] M. Fromme, G. Hoffmann-Schulz, and E. Litvinenko. BEAN – A New Standard Program for Data Analysis at BER-II. In *Real Time Conference, 1999. Santa Fe 1999. 11th IEEE NPSS*, pages 354–358, 1999. 41

[fis] The FishEye Project Page - Real-Time Notifications, web-bases reporting, visualization and code sharing [online]. URL: http://www.atlassian.com/software/fisheye/ [cited 2011-09-17]. 154

[Fit] Michael Fitzpatrick. Using neutron diffraction for the solution of engineering problems: a perspective from academia and industry in the UK. URL: http://neutrons.ornl.gov/workshops/nst2/presentations/05-fitzpatrick.pdf [cited 2009-08-21]. 39

[FJS+08] S.A. Flemming, J.A. James, R. Schneider, R.C. Wimpory, M. Hofmann, M. Schobert, C. Randau, F. Holstein, and T. Weiss. The Open Inspire Architecture for Control, Data Reduction and Analysis. *NOBUGS 2008*, 10 2008. URL:

http://www.nbi.ansto.gov.au/nobugs2008/papers/paper134.pdf. 13

[FL03] M.E. Fitzpatrick and A. Lodini. *Analysis of residual stress by diffraction using neutron and synchrotron radiation.* CRC, 2003. 21

[Fle05] S. A. Flemming. Entwicklung eines verteilten Client-Server-Systems zur vollautomatischen Instrumentenausrichtung eines Neutronendiffraktometers zur Eigenspannungsanalyse. Master's thesis, TFH-Berlin, 10 2005. 31, 33, 94, 211, 245, 248, 249, 251

[Fow00] M. Fowler. POJO [online]. 2000. URL: http://martinfowler.com/bliki/POJO.html. 96

[Fow04] M. Fowler. Inversion of Control Containers and the Dependency Injection Pattern [online]. 2004. URL: http://www.itu.dk/courses/VOP/E2006/8_injection.pdf. 78, 111, 112

[Fow08] Ronald Fowler. Mantid Workspace Storage in NeXus Format. 2008. URL: http://www.nexusformat.org/images/b/bd/MantidNexusWorkspaceFormat.pdf [cited 2009-08-25]. 43

[FPTV92] B.P. Flannery, W.H. Press, S.A. Teukolsky, and W. Vetterling. Numerical recipes in c. *Press Syndicate of the University of Cambridge, New York*, 1992. 248

[Fri06] J. Friedl. *Mastering regular expressions.* O'Reilly Media, Inc., 2006. 119

[Fro] M. Fromme. Software Technique in VITESS. 47

[GBS+07] Andy Götz, Jaro Butanowicz, Lukasz Slezak, Claudio Scafuri, and Giulio Gaio. Ubiquitous TANGO. In *ICALEPCS 2007.* ESRF, Elettra, 2007. URL: http://www.tango-controls.org/Documents/papers/WPPA28.pdf [cited 2009-08-03]. 33

[GDVL+08] Y.P. Guiavarc'h, E. Dintwa, A.M. Van Loey, F.T. Zuber, and M.E. Hendrickx. Validation and use of an enzymic time-temperature integrator to monitor thermal impacts inside a solid/liquid model food. 2008. 46

[GFKW07] L. Giller, U. Filges, G. Kühne, and M. Wohlmuther. Validation of Monte-Carlo simulations with measurements at the ICON beam-line at SINQ. *Nuclear Inst. and Methods in Physics Research, A*, 2007. 46, 48

[GH] A. Götz and N. Hauser. Grand Unified Model for Control and Analysis Systems. URL: `http://lns00.psi.ch/nobugs2004/papers/paper00125.pdf` [cited 2009-08-12]. 37, 87

[GKM+09] A. Götz, W-D. Klotz, P. Maekijaervi, J. Meyer, E. Taurel, and J. Quick. TACO: An object oriented system for PC's running Linux, Windows/NT, OS-9, LynxOS or VxWorks, 2009. 33, 34

[GM] Mark L. Green and Stephen D. Miller. Multitier Portal Architecture for Thin- and Thick-client Neutron Scattering Experiment Support. URL: `gumtree`. 111

[GM07] M. L. Green and S. D. Miller. Multitier Portal Architecture for Thin-and Thick-client Neutron Scattering Experiment Support. In *International Workshop on Grid Computing Environments*, 2007. URL: `http://library.rit.edu/oajournals/index.php/gce/article/viewFile/102/80` [cited 2009-08-11]. 37, 42

[goo11] The Google Trends Engine [online]. 05 2011. URL: `http://www.google.de/trends` [cited 2011-05-29]. 170

[GOP07] R. Gilles, A. Ostermann, and W. Petry. Monte Carlo simulations of the new small-angle neutron scattering instrument SANS-1 at the Heinz Maier-Leibnitz Forschungsneutronenquelle. *Journal of Applied Crystallography*, 40:s428–s432, 2007. 46

[GPZ05] T. Gu, H.K. Pung, and DQ Zhang. Toward an OSGi-based Infrastructure for Context-Aware Applications. *Pervasive Computing, IEEE*, 3(4):66–74, 2005. 86

[Gra07] G.E. Granroth. Fast Monte Carlo simulation of a dispersive sample on the SEQUOIA spectrometer at the SNS. *Journal of Neutron Research*, 15(1):91–94, 2007. 46

[gre] Truly agile project management with GreenHopper for JIRA [online]. URL: http://www.atlassian.com/software/greenhopper/ [cited 2011-09-17]. 154

[GS96] W.R. Gilks and DJ Spiegelhalter. *Markov chain Monte Carlo in practice.* Chapman & Hall/CRC, 1996. 44

[GSN06] E. P. Gilbert, J. C. Schulz, and T. J. Noakes. Quokka -The small-angle neutron scattering instrument at OPAL. *Physica B: Physics of Condensed Matter*, 385:1180–1182, 2006. 37

[GTP+03] A. Götz, E. Taurel, J. L. Pons, P. Verdier, J. M. Chaize, J. Meyer, F.Poncet, G.Heunen, and E.Götz. TANGO a CORBA based Control System. In *ICALEPCS 2003*, ESRF, 6 rue Jules Horowitz, 38043 Grenoble, FRANCE A.Buteau, N.Leclercq, M.Ounsy, Synchrotron Soleil, Saint-Aubin, BP 48, 91192 Gif-sur-Yvette, FRANCE, 2003. ESRF. URL: http://www.tango-controls.org/Documents/papers/icalepcs2003.pdf/ [cited 2009-08-03]. 33, 111

[gum] GumTree - Website [online]. URL: http://gumtree.codehaus.org/ [cited 2009-08-12]. 37

[HCG+] J. Hatje, M. Clausen, C. Gerke, M. Moeller, and H. Rickens. Control System Studio (CSS). In *Proceedings of ICALEPCS07, Knoxville, Tennessee, USA.* URL: https://accelconf.web.cern.ch/accelconf/ica07/PAPERS/MOPB03.PDF [cited 2009-08-17]. 39

[hdfa] HDF vs. HDF5 [online]. URL: http://www.hdfgroup.org/h4vsh5.html [cited 2009-09-22]. 52

[hdfb] The HDF Group - Information, Support, and Software [online]. URL: http://www.hdfgroup.org/ [cited 2012-06-11]. 192

[Hen04] M. Henning. A new Approach to Object-Oriented Middleware. *Internet Computing, IEEE*, 8(1):66–75, 2004. 176, 263

[HG] N. Hauser and A. Götz. Choosing an instrument control system for ANSTO. URL: http://lns00.psi.ch/nobugs2004/papers/paper00092.pdf

[cited 2009-08-11]. 37

[HHS03] D. Hohlwein, J.-U. Hoffmann, and R Schneider. Magnetic interaction parameters from paramagnetic diffuse neutron scattering in MnO. *Physical Review B (Condensed Matter and Materials Physics)*, 68(14):pp. 140408–1–4, 2003. 43

[Hil95] JO Hill. A Server Level API for EPICS. In *Proc. ICALEPCS*, pages 136–141, 1995. URL: `http://www-bd.fnal.gov/icalepcs/abstracts/PDF/w1ab.pdf` [cited 2009-08-05]. 34

[Hil08] B.M. Hill. Free Software Project Management HOWTO. *Retrieved July*, 3:2009, 2008. 90

[HK92] M.T. Hutchings and A.D. Krawitz. *Measurement of residual and applied stress using neutron diffraction.* Springer, 1992. 21

[HKM97] H. Heer, M. Könnecke, and D. Maden. The SINQ instrument control software system. *Physica B: Physics of Condensed Matter*, 241:124–126, 1997. 36

[HL04] E. Hatcher and S. Loughran. *Java-Entwicklung mit Ant.* Mitp, 2004. 155, 161

[HLH+04] P. V. Hathaway, T. Lam, N. Hauser, A. Götz, and F. Franceschini. An Agile Approach to Instrument Control Software at the Australian Nuclear Science and Technology Organisation. 2004. URL: `http://lns00.psi.ch/nobugs2004/papers/paper00091.pdf` [cited 2009-08-12]. 37

[HO07] M. Haft and B. Olleck. Komponentenbasierte Client-Architektur. *Informatik-Spektrum*, 30(3):143–158, 2007. 112

[HSG+07] M. Hoelzel, A. Senyshyn, R. Gilles, H. Boysen, and H. Fuess. Scientific Review: The Structure Powder Diffractometer SPODI. *Neutron News*, 18(4):23–26, 2007. 46

[Hut05] M.T. Hutchings. *Introduction to the characterization of residual stress by neutron diffraction.* CRC Press, 2005. 23

[hzb] Helmholtz Zentrum Berlin für Materialien und Energie [online]. URL: http://www.helmholtz-berlin.de/ [cited 2009-08-05]. 35

[iae] Data Acquisition & Analysis for Neutron Beam Line Experiments. URL: http://www-naweb.iaea.org/napc/physics/research_reactors/meetings/Final_Report_08CT14319_acq.pdf [cited 2009-08-23]. 40

[idl] IDL - ITT Visual Information Solutions, Image Processing and Data Analysis [online]. URL: http://www.ittvis.com/ [cited 2009-08-22]. 39

[ife] IFE - Institute for Energy Technology [online]. URL: http://www.ife.no/laboratories/neutron_activation_lab/index_html-en?set_language=en&cl=en [cited 2009-09-15]. 46

[ill] ILL :: Neutrons for science : www [online]. URL: http://www.ill.eu/ [cited 2009-09-15]. 46

[int11] IntelliJ IDEA Project Webpage [online]. 05 2011. URL: http://www.jetbrains.com/idea/ [cited 2011-05-29]. 169

[ipn] Intense Pulsed Neutron Source Home Page [online]. URL: http://www.pns.anl.gov/ [cited 2009-08-23]. 42

[isa] IPNS - Computing - Web Project (ISAW) [online]. URL: http://www.pns.anl.gov/computing/isaw/index.shtml [cited 2009-08-23]. 42

[isi] ISIS Pulsed Neutron and Muon Source [online]. URL: http://www.isis.rl.ac.uk/ [cited 2009-08-22]. 39, 42

[jap12] Japinotes - The NeXus API for Java [online]. 2012. URL: http://wiki.nexusformat.org/index.php?title=Japinotes&oldid=4438 [cited 2012-06-16]. 193

[Jar] JAR File Specification [online]. URL: http://download.oracle.com/javase/1.5.0/docs/guide/jar/jar.html [cited 2010-11-11]. 91, 92

[jde11] JDeveloper Project Webpage - A Java IDE [online]. 05 2011. URL: http://www.oracle.com/technetwork/developer-tools/jdev/overview/index.html [cited 2011-05-29]. 169

[jdk] jdk7: Java SE 7 [online]. URL: https://jdk7.dev.java.net/ [cited 2010-10-31]. 86

[jdo] JDOM - Java DOM XML Parser [online]. URL: http://www.jdom.org/index.html [cited 2011-05-19]. 129

[JE07] J.A. James and L. Edwards. Application of robot kinematics methods to the simulation and control of neutron beam line positioning systems. *Nuclear Inst. and Methods in Physics Research, A*, 571(3):709–718, 2007. 40

[JEW97] MW Johnson, L. Edwards, and PJ Withers. ENGIN—A new instrument for engineers. *Physica B: Physics of Condensed Matter*, 234:1141–1143, 1997. 39

[JHA+] R. Johnson, J. Hoeller, A. Arendsen, C. Sampaleanu, R. Harrop, T. Risberg, et al. The Spring Framework-Reference Documentation. 112

[jig] OpenJDK: Project Jigsaw [online]. URL: http://openjdk.java.net/projects/jigsaw/ [cited 2010-10-31]. 86

[jin] JINR - Joint Institute For Nuclear Research - Dubna [online]. URL: http://www.jinr.ru/ [cited 2009-09-15]. 46

[jir] Bug, Issue and Project Tracking for Software Development - JIRA [online]. URL: http://www.atlassian.com/software/jira/ [cited 2011-09-17]. 154

[JJ06] G.W. Johnson and R. Jennings. *LabVIEW graphical programming.* McGraw-Hill Professional, 2006. 41

[JN] J.P.Lewis and Ulrich Neumann. Performance of java versus c++ [online]. URL: http://www.idiom.com/~zilla/Computer/javaCbenchmark.html [cited 2010-10-25]. 82

[JOP+08] JA James, EC Oliver, A. Paradowska, C.R. Hubbard, J. Schmidlin, and L. Edwards. Robotics Methods for Beam Line Instrument Simulation and Control. 2008. URL: http://www.nbi.ansto.gov.au/nobugs2008/papers/paper107.pdf [cited 2009-08-21]. 40

[jsc] Scripting for the java platform [online]. URL: https://scripting.dev.java.net/. 81

[JSDE02] JA James, JR Santistiban, MR Daymond, and L. Edwards. Use of a Virtual Laboratory to plan, execute and analyse Neutron Strain Scanning experiments. *Arxiv preprint cond-mat/0210432*, 2002. URL: http://arxiv.org/pdf/cond-mat/0210432 [cited 2009-08-21]. 39

[JSED04] JA James, JR Santisteban, L. Edwards, and MR Daymond. A virtual laboratory for neutron and synchrotron strain scanning. *Physica B: Physics of Condensed Matter*, 350(1-3S):743–746, 2004. 39

[jsra] [online]URL: ttp://jcp.org/en/jsr/detail?id=294 [cited 2010-10-31]. 86

[jsrb] The Java Community Process(SM) Program - JSRs: Java Specification Requests - JSR 175 [online]. URL: http://www.jcp.org/en/jsr/detail?id=175 [cited 20. Nov. 2010]. 98

[KA01] A.D. Krawitz and K.R. ABRAMS. Introduction to diffraction in materials science and engineering. *New York*, 2001. 21

[KAEJ98] W-D. Klotz, A.Götz, E.Taurel, and J.Meyer. TANGO - object oriented device control implemented in CORBA and DCOM. 07 1998. 33

[KAJ+97a] M. R. Kraimer, J. Anderson, A. Johnson, E. Norum, and J. Hill. EPICS: Input/Output Controller Application Developer's Guide. *Argonne National Laboratory APS, July*, 1997. URL: http://www.aps.anl.gov/epics/base/R3-14/9-docs/AppDevGuide.pdf [cited 2009-08-05]. 34

[KAJ+97b] M. R. Kraimer, J. Anderson, A. Johnson, E. Norum, and J. Hill. EPICS: Input/Output Controller Application Developer's Guide. *Argonne National Labo-*

ratory APS, July, 1997. URL: http://www.aps.anl.gov/epics/base/R3-14/9-docs/AppDevGuide.pdf [cited 2009-08-05]. 35

[Kas] K. Kasemir. Control System Studio (CSS) Data Browser. URL: http://accelconf.web.cern.ch/AccelConf/pc08/papers/tup009.pdf [cited 2009-08-17]. 39

[Kas07] K. Kasemir. Control System Studio Applications. In *ICALEPCS*, 2007. URL: http://accelconf.web.cern.ch/accelconf/ica07/PAPERS/ROPB02.PDF [cited 2009-08-17]. 39

[KBB98] Will Kleber, H.-J. Bautsch, and Joachim Bohm. *Einführung in die Kristallographie.* Verlag Technik Berlin, 18. auflage edition, 1998. 20

[KBC+] I. Kriznar, J. Bobnar, L.M.R.C. Cosylab, P. Duval, W.H. Gong, and H. DESY. Beyond Abeans. URL: http://accelconf.web.cern.ch/accelconf/ica07/PAPERS/TPPA24.PDF [cited 2009-08-17]. 38, 39

[KBF+00] I. Kaiser, H. Boysen, F. Frey, M. Lerch, D. Hohlwein, and R. Schneider. Diffuse scattering in quaternary single crystals in the system Zr-Y-O-N. *Zeitschrift fur Kristallographie*, 215(8):pp. 437–40, 2000. 43

[KLM+02] SN Klausen, K. Lefmann, DF McMorrow, F. Altorfer, S. Janssen, and M. Lüthy. Simulations and experiments on RITA-2 at PSI. *Applied Physics A: Materials Science & Processing*, 74:1508–1510, 2002. 46

[KN02] Jens Krüger and Jürgen Neuhaus. Instrument Control with TACO at the FRM-II. In *NOBUGS2002 Conference*, 2002. 34

[KPP+] I. Kriznar, G. Pajor, M. Plesko, M. Sekoranja, G. Tkacik, D. Vitas, K. Cerff, and W. Mexner. The Upgrade of the ANKA Control System to ACS (Advanced Control System). *Proceedings of ICALEPCS2003, Gyeongju, Korea*, page 45. URL: https://accelconf.web.cern.ch/accelconf/ica03/PAPERS/WP541.PDF [cited 2009-08-20]. 38

[Kra01] A.D. Krawitz. Neutron stress measurements in the 21st century. *Journal of Neutron Research*, 9(2):119–127, 2001. 21

[Kra08] A Krawitz. The Early History of Neutron Stress Measurements. *Materials Science Forum, 2008*, Vols. 571-572 pp 3-11., 2008. 21

[KW00] J. Kohlbrecher and W. Wagner. The new SANS instrument at the Swiss spallation source SINQ. *Journal of Applied Crystallography*, 33(3):804–806, 2000. 36

[KZH+08] Mark Konnecke, Markus Zolliker, Nick Hauser, Ferdi Franceshcini, and Tony Lam. Treepath Based Instrument Control. In *NOBUGS 2008*, 2008. URL: `http://www.nbi.ansto.gov.au/nobugs2008/papers/paper132.pdf` [cited 2009-08-18]. 37

[lab] LabVIEW Zone - Developer Zone - National Instruments [online]. URL: `http://zone.ni.com/devzone/cda/tut/p/id/5053` [cited 2009-08-24]. 41

[lab09] NI LabVIEW - The Software That Powers Virtual Instrumentation - Products and Services - National Instruments [online]. 08 2009. URL: `http://www.ni.com/labview/` [cited 2009-08-24]. 41

[lan] Los Alamos National Lab [online]. URL: `http://lanl.gov/` [cited 2009-09-15]. 48

[LCBM05] M. Langheinrich, V. Coroama, J. Bohn, and F. Mattern. Living in a smart Environment – Implications for the coming ubiquitous information society. *Telecommunications Review*, 15(1):132–143, 2005. 111

[Lew00] S. A. Lewis. Overview of the Experimental Physics and Industrial Control System: EPICS. 2000. URL: `http://epics.org/overview.pdf` [cited 2009-08-05]. 34

[LGFH06] T. Lam, A. Götz, F. Franceschini, and N. Hauser. GumTree - A Java based GUI framework for beamline experiments. *Journal of Neutron Research*, 14(2):167–175, 2006. URL: `http://lns00.psi.ch/nobugs2004/papers/paper00090.pdf`

[cited 2009-08-11]. 37

[LGWM05] K. Lieutenant, T. Gutberlet, A. Wiedenmann, and F. Mezei. Monte-Carlo simulations of small angle neutron scattering instruments at European spallation source. *Nuclear Inst. and Methods in Physics Research, A*, 553(3):592–603, 2005. 47

[LHG+06] T. Lam, N. Hauser, A. Götz, P. Hathaway, F. Franceschini, H. Rayner, and L. Zhang. GumTree - An integrated scientific experiment environment. *Physica B: Physics of Condensed Matter*, 385:1330–1332, 2006. 37

[LKH06] Tony Lam, Darren Kelly, and Nick Hauser. NeXus and Collaboration DANSE kick-off meeting. 08 2006. URL: `http://www.cacr.caltech.edu/projects/danse/talks/kickoff/13-Hauser/DANSEKickOff.pdf` [cited 2012-06-09]. 191

[LN99] K. Lefmann and K. Nielsen. McStas, a general software package for neutron ray-tracing simulations. *Neutron News*, 10(3):20–23, 1999. 46

[LNTL00] K. Lefmann, K. Nielsen, A. Tennant, and B. Lake. McStas 1.1: a tool for building neutron Monte Carlo simulations. *Physica B: Physics of Condensed Matter*, 276:152–153, 2000. 46

[LO05] R. Love and Safari Tech Books Online. *Linux kernel development*, volume 50. Novell Press USA, 2005. 94

[LW02] W.T. Lee and X.L. Wang. IDEAS: A general purpose software package for simulating neutron scattering instruments. *Neutron News*, 13(4):30–34, 2002. 47

[LWF] K. Lefmann, P.K. Willendrup, and E. Farhi. Component Manual for the Neutron Ray-Tracing Package McStas, Version 1.11. 46

[LWR+02] W.T. Lee, X.L. Wang, JL Robertson, F. Klose, and C. Rehm. IDEAS–A Monte Carlo simulation package for neutron-scattering instrumentation. *Applied Physics A: Materials Science & Processing*, 74:1502–1504, 2002. 47

[LZM+] K. Lieutenant, G. Zsigmond, S. Manoshin, M. Fromme, H.N. Bordallo, D. Champion, J. Peters, F. Mezei, and EUV Lithography. Neutron instrument simulation and optimization using the software package VITESS (Proceedings Paper). 47

[man] The Mantid Application Framework [online]. URL: `http://www.mantidproject.org` [cited 2009-08-25]. 43

[MBF99] M. Mattsson, J. Bosch, and M.E. Fayad. Framework integration problems, causes, solutions. *Communications of the ACM*, 42(10):87, 1999. 111

[mcn] MCNPX Homepage [online]. 48

[mcsa] McStas - A Neutron Ray-Trace Simulation Package [online]. URL: `http://www.mcstas.org/` [cited 2009-09-14]. 45, 46

[mcsb] McStas Publications [online]. URL: `http://www.mcstas.org/documentation/publications/` [cited 2009-09-15]. 46

[mcx] Main Page - McXtraceWiki [online]. URL: `http://www.mcxtrace.org/` [cited 2009-09-15]. 46

[min11] MinGW | Minimalist GNU for Windows [online]. 09 2011. URL: `http://www.mingw.org/` [cited 2011-09-16]. 152

[Mit03] J.C. Mitchell. *Concepts in Programming Languages*. Cambridge Univ Pr, 2003. 34

[MK03] H. Masuhara and K. Kawauchi. Dataflow Pointcut in Aspect-Oriented Programming. *Programming Languages and Systems*, pages 105–121, 2003. 95

[MZA08] S. Mayer, G. Zsigmond, and P. Allenspach. Monte-Carlo simulation of phase space transformation of ultra-cold neutrons. *Nuclear Inst. and Methods in Physics Research, A*, 586(1):110–115, 2008. URL: `http://ucn.web.psi.ch/papers/PSTofUCN_NOP07.pdf`. 47

[nbi] Niels Bohr Institutet – Københavns Universitet Københavns Universitet [online]. URL: http://www.nbi.ku.dk/ [cited 2009-09-15]. 46

[nec] NECSA Website [online]. URL: http://www.necsa.co.za/ [cited 2009-08-06]. 36

[net] Unidata - NetCDF - Network Common Data Format [online]. URL: http://www.unidata.ucar.edu/software/netcdf/ [cited 2012-06-11]. 192

[net11] NetBeans Project Webpage - A Java IDE [online]. 05 2011. URL: http://netbeans.org/ [cited 2011-05-29]. 169, 173, 174

[Neu98] Jürgen Neuhaus. Konzept zur Instrumentsteuerung am FRM-II, 8 1998. 33, 34

[nexa] NeXML: A generative XML Schema with EMF Java bindings for the NeXus Format | Elements of the Web [online]. URL: http://webel.com.au/content/nexml-generative-xml-schema-emf-java-bindings-nexus-format [cited 2012-06-09]. 191

[nexb] NeXus - Instrument Definitions [online]. URL: http://www.nexusformat.org/Instruments [cited 2009-09-22]. 52

[nexc] NeXus Design Overview - Classes [online]. URL: http://www.nexusformat.org/Design [cited 2009-09-22]. 50

[nex09] Introduction to the NeXus Design [online]. 09 2009. URL: http://www.nexusformat.org/Introduction [cited 2009-09-22]. 50

[nia09] NIAC - NeXus [online]. 09 2009. URL: http://www.nexusformat.org/NIAC [cited 2009-09-19]. 49, 191

[nisa] NISP - P.A.Seeger Home Page [online]. 47

[nisb] NIST Center for Neutron Research [online]. URL: http://www.ncnr.nist.gov/ [cited 2009-08-25]. 43

[NKM95] VM Nield, DA Keen, and RL McGreevy. The interpretation of single-crystal diffuse scattering using reverse Monte Carlo modelling. *Acta Crystallographica*

Section A: Foundations of Crystallography, 51(5):763–771, 1995. 44

[NL00] K. Nielsen and K. Lefmann. Monte Carlo simulations of neutron-scattering instruments using McStas. *Physica B: Physics of Condensed Matter*, 283(4):426–432, 2000. 46

[nob] Main Page - NOBUGS Conference [online]. URL: `http://www.nobugsconference.org` [cited 2011-08-20]. 81, 206

[nom] ILL :: Neutrons for science : Leading-edge software: NOMAD [online]. URL: `http://www.ill.eu/science-technology/neutron-technology-at-ill/leading-edge-software-nomad/` [cited 2009-08-23]. 40

[nsi11] The Nullsoft Scriptable Install System Webpage [online]. 09 2011. URL: `http://nsis.sourceforge.net` [cited 2011-09-16]. 152

[nuj11] Nujan: Pure Java NetCDF4 and HDF5 writer [online]. 2011. URL: `http://www.ral.ucar.edu/~steves/nujan.html` [cited 2012-06-11]. 192, 266

[OMG04] OMG. Common Object Request Broker Common Object Request Broker Architecture: Core Specification. Technical report, Object Management Group, 03 2004. URL: `http://www.omg.org/docs/formal/04-03-12.pdf` [cited 2009-06-10]. 176, 207

[ope] OpenGENIE Home Page [online]. URL: `http://www.isis.rl.ac.uk/OpenGENIE/` [cited 2009-08-22]. 42

[osg] OSGi Alliance | Main [online]. URL: `http://www.osgi.org/` [cited 2010-10-31]. 86

[OSJ+04] E. Oliver, J. Santisteban, J. James, M. Daymond, and J. Dann. ENGIN-X user manual. *ISIS, Rutherford Laboratory*, 2004. URL: `http://www.isis.rl.ac.uk/engineering/documentation/enginx_manual.pdf` [cited 2009-08-22]. 39, 42, 202

[Ous94] John K Ousterhout. *Tcl and the Tk toolkit.* Addison-Wesley, 1994. 36

[Owe06] M. Owens. *The definitive guide to SQLite.* Apress, 2006. 200

[PCAE78] H.J. Prask, C.S.C., H.A. Alperin, and E.Prince. Applications of Neutron Diffraction of Non-Destructive Testing Problems. In *Summary of activities. NBS Reactor, July 1976 to June 1977 p. NBS*, number Technical Note 969. (1978), pages 73–76, 1978. 21

[pic11] The Pico Container Website [online]. 2011. URL: `http://picocontainer.org/` [cited 11.02.2011]. 112

[PP05] D. Pilone and N. Pitman. *UML 2.0 in a Nutshell.* O'Reilly, 2005. 14

[psi] Paul Scherrer Institute [online]. URL: `http://www.psi.ch/` [cited 2009-08-05]. 35, 46

[Pur] D. Purcell. Control System Studio and the SNS Relational Database. URL: `http://accelconf.web.cern.ch/AccelConf/pc08/papers/tuz03.pdf` [cited 2009-08-17]. 39

[PZS+02] M. Plesko, K. Zagar, M. Sekoranja, J. Dovc, M. Kadunc, and I. Kriznar. ACS–The advanced control system. In *4th International Workshop on Personal Computers and Particle Accelerator Controls.* Citeseer, 2002. URL: `http://accelconf.web.cern.ch/AccelConf/e98/PAPERS/TUP02E.PDF` [cited 2009-08-20]. 38

[Qui94] D. Quinlan. File System Standard (FSSTD). *Linux Journal*, 1994(2es):6, 1994. 146

[Ran07] Christian Randau. Implementierung und Bewertung des kontinuierlichen neutronendiffraktometrischen Texturanalyse-Experiments. Master's thesis, Fachhochschule für Technik und Wirtschaft Berlin, 08 2007. 204

[RD90] R. Rew and G. Davis. NetCDF: An Interface for Scientific Data Access. *Computer Graphics and Applications, IEEE*, 10(4):76–82, 1990. 192

[rel] Software Release Practice HOWTO [online]. URL: `http://tldp.org/HOWTO/Software-Release-Practice-HOWTO/index.html` [cited 2010-11-09]. 90

[res] RESTRAX Homepage [online]. URL: `http://neutron.ujf.cas.cz/restrax/` [cited 2009-09-14]. 47

[rfc] RFC 822 - Standard for the Format of ARPA Internet Text Messages [online]. URL: `http://www.ietf.org/rfc/rfc0822.txt` [cited 2010-11-11]. 92

[RHH+06] H. Rayner, P. Hathaway, N. Hauser, Y. Fei, F. Franceschini, and T. Lam. GumTree: Data reduction. *Physica B: Physics of Condensed Matter*, 385:1333–1335, 2006. 37, 38

[RI07] A. Radulescu and A. Ioffe. Neutron guide system for small-angle neutron scattering instruments of the Jülich Centre for Neutron Science at the FRM-II. *Nuclear Inst. and Methods in Physics Research, A*, 2007. 45

[ris] Risø DTU National Laboratory for Sustainable Energy. Research in energy technology, energy planning, energy systems of the future and climate technology [online]. URL: `http://www.risoe.dk` [cited 2009-09-15]. 46

[RR09] J. Robertson and S. Robertson. Volere Requirements Specification Template -Edition 14 (January 2009), 01 2009. 55

[RS07] Lutz Rossa and Olaf-Peter Sauer. CARESS - Die HMI Software für Neutronenstreuexperimente. In *Studiengruppe fuer Elektronische Instrumentierung*, pages 170–179. Dr. F. Wulf, 07 2007. 32, 207, 209

[SAF00] A.P. Sage, J.E. Armstrong, and Knovel (Firm). *Introduction to Systems Engineering*. Wiley, 2000. 55

[SAK95] P. Stanley, J. Anderson, and M. Kraimer. EPICS record reference manual. *LANL, APS/ANL*, 1995. 35

[Sau07] Olaf-Peter Sauer. CARESS - Die HMI Software fuer Neutronenstreuexperimente. In *Studiengruppe fuer Elektronische Instrumentierung*, pages 180–182. Dr. F. Wulf, 07 2007. 32, 207

[SB78] S. Steenstrup and B. Buras. A Monte Carlo simulation of ultra-cold neutron production by Bragg reflection from a moving single crystal. *Nucl. Instrum.*

Methods, 154:549–55, 1978. 44

[SBFH04] C. Schanzer, P. Böni, U. Filges, and T. Hils. Advanced geometries for ballistic neutron guides. *Nuclear Inst. and Methods in Physics Research, A*, 529(1-3):63–68, 2004. 46

[Sch97] R.R. Schaller. Moore's law: Past, Present and Future. *Spectrum, IEEE*, 34(6):52–59, 1997. 67

[Sch03] J. Schmuller. Jetzt lerne ich UML: Der einfache Einstieg in die visuelle Objektmodellierung. *Markt+ Technik*, 2003. 14

[SD04] P.A. Seeger and L.L. Daemen. The neutron instrument simulation package, NISP. In *Proceedings SPIE*, volume 5536, pages 109–123, 2004. 47

[SDF+02] PA Seeger, LL Daemen, E. Farhi, W.T. Lee, X.L. Wang, L. Passell, J. Šaroun, and G. Zsigmond. Monte carlo code comparisons for a model instrument. *Neutron News*, 13(4):24–29, 2002. 45, 47

[SDJE06] JR Santisteban, MR Daymond, JA James, and L. Edwards. ENGIN-X: A third-generation neutron strain scanner. *Journal of Applied Crystallography*, 39(6):812–825, 2006. 39, 202

[SDTH00] PA Seeger, LL Daemen, TG Thelliez, and RP Hjelm. Neutron instrument simulations in the next millennium. *Physica B: Physics of Condensed Matter*, 283(4):433–435, 2000. 47

[See95] PA Seeger. The MCLIB Library: Monte Carlo simulation of neutron scattering instruments. In *Conference: ICANS-XIII: 13. international collaboration on advanced neutron sources, Villigen (Switzerland), 11-14 Oct 1995*, 1995. 47

[SFS04] J. Schwarz, A. Farris, and H. Sommer. The ALMA software architecture. In *Proceedings of SPIE*, volume 5496, pages 190–204, 2004. URL: `http://www.eso.org/~hsommer/work/publications/SPIE2004-5496-22_paper.pdf` [cited 2009-08-27]. 38

[sic] LNS Computing WWW-Page [online]. URL: http://lns00.psi.ch/ [cited 2009-08-06]. 36

[sin] Swiss Spallation Neutron Source SINQ [online]. URL: http://sinq.web.psi.ch/ [cited 2009-08-06]. 36

[ŠK06] J. Šaroun and J. Kulda. MC ray-tracing optimization of lobster-eye focusing devices with RESTRAX. *Physica B: Physics of Condensed Matter*, 385:1250–1252, 2006. 47

[sol09] Synchrotron SOLEIL - Accueil [online]. 2009. URL: http://www.synchrotron-soleil.fr/ [cited 2009-08-02]. 34

[spe] CSS / : Certified Scientific Software [online]. URL: http://www.certif.com/ [cited 2009-08-23]. 40

[spi99] Proceedings of SPIE; Monte Carlo tool for neutron optics and neutron scattering instrument design. 3771(1):80–89, 1999. doi:10.1117/12.363709. 47

[SS07] S. Sumathi and P. Surekha. *LabVIEW based Advanced Instrumentation Systems.* Springer-Verlag Berlin Heidelberg, 2007. 41

[sub11] Subrepository - Mercurial - Version Control System [online]. 08 2011. URL: http://mercurial.selenic.com/wiki/Subrepository [cited 2011-08-20]. 85

[Sun07] K. Sinha Sunil. Complementary use of neutron and synchrotron X-ray scattering for problems of interest in condensed matter physics. *Journal of Physics and Chemistry of Solids*, 68:2048–2051, August 2007. 20

[tan09] The TANGO Controls Website [online]. 2009. URL: http://www.tango-controls.org/ [cited 2009-08-01]. 33

[TBO03] D. Thompson, W. Blokland, and O. R. ORNL. A Shared Memory Interface between LabVIEW and EPICS. In *ICALEPCS*, pages 13–17, 2003. URL: https://wiki.gsi.de/pub/Epics/ConnectingLabVIEWandEPICS/TU514.PDF [cited 2009-08-05]. 34

[TBW+06] J. Tao, CJ Benmore, TG Worlton, JM Carpenter, D. Mikkelson, R. Mikkelson, J. Siewenie, J. Hammonds, and A. Chatterjee. Time-of-flight neutron total scattering data analysis implemented in the software suite ISAW. *uclear Inst. and Methods in Physics Research, A*, 562(1):422–432, 2006. 42

[Tea02] C.P. Team. Capability Maturity Model Integration (CMMI), Version 1.1 – Continuous Representation. 2002. 57

[TK06] Jeffrey Travis and Jim Kring. *LabVIEW for Everyone: Graphical Programming Made Easy and Fun, Third Edition.* Prentice Hall, 2006. 41

[tvn] Main Page - Experiment - TVnexus [online]. URL: `https://wiki.helmholtz-berlin.de/experiment/index.php/Main_Page` [cited 2009-08-25]. 44

[typa] Sites made with TYPO3 [online]. URL: `http://typo3.org/about/sites-made-with-typo3/` [cited 2011-08-27]. 148

[typb] TYPO3 - The Enterprise Open Source CMS [online]. URL: `http://typo3.org/` [cited 2011-08-27]. 148

[uml08] OMG Unified Modeling LanguageTM (OMG UML), Infrastructure. Technical report, Object Management Group, http://www.omg.org/docs/ptc/08-05-04.pdf, 2008. 14

[Unr02] T. Unruh. Instrument control at FRM II using TACO and NICOS. *Proceedings of NOBUGS*, 2002. 33, 34

[UO08] O. Uca and C. Ohms. Monte Carlo simulations of the SANS instrument in Petten. *Physica B: Physics of Condensed Matter*, 403(21-22):3892–3895, 2008. 45

[VdHW03] A. Van der Hoek and A.L. Wolf. Software release management for component-based software. *Software: Practice and Experience*, 33(1):77–98, 2003. 90

[VGF+03] G. Venturi, E. Guarini, F. Formisano, A. Orecchini, A. Cunsolo, C. Petrillo, F. Sacchetti, and F. Barocchi. Optimizing the Setup of the BRISP Spectrometer

by Upgraded McStas Simulations. *Journal of Neutron Research*, 11(3):165–178, 2003. 46

[vita] VITESS [online]. URL: `http://www.helmholtz-berlin.de/forschung/grossgeraete/neutronenstreuung/projekte/vitess/index_de.html` [cited 2009-09-14]. 46, 47

[vitb] VMEbus International Trade Association (VITA) [online]. URL: `http://www.vita.com/` [cited 2009-10-05]. 30

[VKK+] I. Verstovsek, M. Kadunc, J. Kamenik, I. Kriznar, G. Pajor, M. Plesko, A. Pucelj, M. Sekoranja, G. Tkacik, and D. Vitas. Abeans: Application Development Framework for Java. In *Proceedings of ICALEPCS2003, Gyeongju, Korea*. URL: `https://accelconf.web.cern.ch/accelconf/ica03/PAPERS/WE203.PDF` [cited 2009-08-17]. 38

[WFL04] P. Willendrup, E. Farhi, and K. Lefmann. McStas 1.7-a new version of the flexible Monte Carlo neutron scattering package. *Physica B: Physics of Condensed Matter*, 350(1-3S):735–737, 2004. 46

[WFL+05] P. Willendrup, E. Farhi, K. Lefmann, K. Lieutenant, and K. Nielsen. User and Programmers Guide to the Neutron Ray-Tracing Package McStas, Version 1.9. *Risoe Report*, 2005. 46

[WH04] HR Wenk and P.V. Houtte. Texture and anisotropy. *Reports on Progress in Physics*, 67(8):1367–1428, 2004. 23

[WKF+] P. Willendrup, E. Knudsen, E. Farhi, K. Lefmann, DTU Risø, and D. Roskilde. User and Programmers Guide to the Neutron Ray-Tracing Package McStas, Version 1.12. 46

[WMM+04] TG Worlton, DJ Mikkelson, R. Mikkelson, JP Hammonds, J. Tao, and A. Chatterjee. New Developments in Data Analysis at IPNS. *NOBUGS 2004*, 2004. 42

[WSF+00] AR Wildes, J. Saroun, E. Farhi, I. Anderson, P. Høghøj, and A. Brochier. A comparison of Monte-Carlo simulations using RESTRAX and McSTAS with

experiment on IN14. *Physica B: Physics of Condensed Matter*, 276:177–178, 2000. 45, 47

[WSF+02] AR Wildes, J. Saroun, E. Farhi, I. Anderson, P. Hoghoj, and A. Brochier. A comparison of Monte-Carlo simulation programs with experiment: the effect of a focusing guide on resolution. *Applied Physics A: Materials Science & Processing*, 74:1452–1454, 2002. 45, 47

[WZS+00] D. Wechsler, G. Zsigmond, F. Streffer, JA Stride, and F. Mezei. Monte-Carlo simulations for instrumentation at pulsed and continuous sources. *Physica B: Physics of Condensed Matter*, 276:71–72, 2000. 47

[WZSM00] D. Wechsler, G. Zsigmond, F. Streffer, and F. Mezei. VITESS: Virtual instrumentation tool for pulsed and continuous sources. *Neutron News*, 11(4):25–28, 2000. 47

[Yer03] F. Yergeau. UTF-8, A transformation format of ISO 10646. IETF. Technical report, STD 63, RFC 3629 (Standards Track), IETF, 2003. 114

[ZGA+09] S.Y. Zhang, E. Godfrey, B. Abbey, P. Xu, Y. Tomota, D. Liljedahl, O. Zanellato, M. Fitzpatrick, J. Kelleher, S. Siano, et al. Materials Structure and Strain Analysis Using Time-of-flight Neutron Diffraction. In *Proceedings of the World Congress on Engineering*, volume 2, 2009. URL: `http://www.iaeng.org/publication/WCE2009/WCE2009_pp1412-1419.pdf` [cited 2009-08-21]. 39

[zip] ZIP MIME Type - IANA [online]. URL: `http://www.iana.org/assignments/media-types/application/zip` [cited 2010-11-10]. 90

[ZLM02] G. Zsigmond, K. Lieutenant, and F. Mezei. Monte Carlo simulations of neutron scattering instruments by VITESS: Virtual instrumentation tool for ESS. *Neutron News*, 13(4):11–14, 2002. 47

[ZLM+04] G. Zsigmond, K. Lieutenant, S. Manoshin, HN Bordallo, J.D.M. Champion, J. Peters, JM Carpenter, and F. Mezei. A survey of simulations of complex neutronic systems by VITESS. *Nuclear Inst. and Methods in Physics Research,*

A, 529(1-3):218–222, 2004. 47

[ZSK+06] K. Zeitelhack, C. Schanzer, A. Kastenmüller, A. Röhrmoser, C. Daniel, J. Franke, E. Gutsmiedl, V. Kudryashov, D. Maier, D. Päthe, et al. Measurement of neutron flux and beam divergence at the cold neutron guide system of the new Munich research reactor FRM-II. *Nuclear Inst. and Methods in Physics Research, A*, 560(2):444–453, 2006. 46

LIST OF FIGURES

LIST OF TABLES

ACRONYMS

ACS ALMA Common Software. 38

ACS Advanced Control System. 38

ADET Area Detector. 29

ANL Argonne National Laboratory. 35

ANSTO Australian Nuclear Science and Technology Organisation. 2, 36, 37, 40, 47, 56, 59, 212, 263

API Application Programming Interface. 34, 35, 49–51, 53, 69, 106, 110, 136, 138, 154, 166–169, 191, 197–200, 219, 222, 229, 276, 303

APT Advanced Packaging Tool. 157

BENSC Berlin Neutron Scattering Center. 42, 44, 214, 215

BER-II Berliner Experimentier Reaktor - II. 24, 49, 305

BESSY Berliner Elektronenspeicherring-Gesellschaft für Synchrotronstrahlung m.b.H.. 56

BHT Beuth University for Applied Sciences Berlin. 152

BNL Brookhaven National Laboratory of the NIST. 48

CAD Computer Aided Design. 248

CAMAC Computer Automated Measurement And Control. 30, 31

CCD Charge-coupled Device. 253

CIS Continuous Integration System. 154–156, 173

CLI Command Line Interface. 54, 144, 147, 149, 187–189, 215–217, 245

CMMI Capability Maturity Model Integration. 57

CMS Content Management System. 152

CORBA Common Object Request Broker Architecture. 33, 97, 98, 182, 215, 218, 219, 221–223, 228, 232, 274

CPU Central Processing Unit. 136

CSS Control System Studio. 39

CVB Common Vision Blox. 253, 256

DAC Data Acquisition and Control. 58, 74, 76–80, 108

DAL Data Access Layer. 39

DANSE Data Analysis for Neutron Scattering Experiments. 35, 36, 54, 97

DAVE Data Analysis and Visualization Environment. 43

DESY Deutsches Elektronen-Synchrotron. 35, 39, 56, 59

DI Dependency Injection. 80, 114, 115

DLL Dynamic Link Library. 48

DRA Data Reduction and Analysis Module. 38

DTD Document Type Definition. 198

E-Hall Experimental Hall. 24, 25

EAR Enterprise Application Archive. 94

EJB Enterprise JavaBean. 88, 98, 115

EPICS Experimental Physics and Industrial Control System. 35, 38, 39, 77, 243

ESO European Southern Observatory. 38

ESS European Spallation Source. 47

exe Windows Executable. 157

FAQ Frequently Asked Question. 311

FHS Filesystem Hierarchy Standard. 150

FPGA Field Programmable Gate Array. 76

FRM-II Research-Neutronsource Heinz Maier-Leibnitz. 2, 24, 33, 44, 47, 48, 59, 212, 214, 215, 263, 305

FS File System. 144, 145

GPL GNU General Public License. 46

GUI Graphical User Interface. 40, 43, 47, 48, 60, 120, 121, 130, 146, 147, 174, 175, 179, 181, 187, 189, 191, 196, 204, 210, 213, 215, 216, 251, 256, 262, 276

GUM Grand Unified Model for Control and Analysis Systems. 38, 89

GumTree Graphical User interface for Multiple and Time Resolved Experiments. 37–39, 43, 54

HDF Hierarchical Data Format. 53, 198, 199, 202

HDF5 Hierarchical Data Format Version 5. 40, 276

HMI Hahn Meitner Institute. 56

HMI Human Machine Interface. 196, 197

HTML Hypertext Markup Language. 125, 274

HZB Helmholtz-Centre Berlin for Materials and Energy. 2, 24, 30, 32, 33, 35, 44, 47, 48, 56, 59, 63, 64, 66, 68, 110, 152, 185, 212, 214, 217, 218, 263, 300, 305

HZDR Helmholtz-Zentrum Dresden-Rossendorf. 59

IAEA International Atomic Energy Agency. 320

IDE Integrated Development Environment. 12, 74, 84, 86, 161, 162, 173–175, 178, 212, 271, 272, 275

IDL Interface Definition Language. 223

IDL Interactive Data Language. 40, 43, 46

idlj Interface Definition Language to Java Compiler. 228, 232

IEEE 1233 Guide for Developing System Requirements Specifications. 55

IEEE 830 Recommended Practice for Software Requirements Specifications. 55

IFE Institute for Energy Technology - Norway. 47

ILL Institute Laue-Langevin. 21, 40, 46, 47, 56, 59

IoC Inversion of Control. 80, 114–116, 265, 266

IP Internet Protocol. 60, 150

IPNS Intense Pulsed Neutron Source. 43

ISAW Integrated Spectral Analysis Workbench. 43, 54

IT Information Technology. 2, 13

JAR Java Archive. 94, 96, 103–106, 164, 173

JDBC Java Database Connectivity. 49

JDK Java Development Kit. 160, 162, 169, 198, 228

JEE Java Enterprise Edition. 88

JINR Joint Institute For Nuclear Research - Dubna. 47

JIT Just in Time. 84

JNI Java Native Interface. 53, 106, 135

JSR 203 JSR 203: More new I/O APIs for the Java platform (NIO.2). 169

JSR 220 JSR 220 : Enterprise JavaBeans 3.0. 88

JSR 223 JSR 223 : Scription for the Java Platform. 83, 84, 212, 272

JSR 294 JSR 294 : Improved Modularity Support in the Java Programming Language. 88

JVM Java Virtual Machine. 136, 137, 145–147, 183, 198

LabVIEW Laboratory Virtual Instrumentation Engineering Workbench. 41, 42

LANL Los Alamos National Lab. 48

LnF Look and Feel. 60, 150, 189, 282

Mantid Manipulation and Analysis Toolkit for ISIS Data. 43, 44, 54

MCNPX Monte Carlo N-Particle eXtended. 48

McStas Monte Carlo Simulation of Triple Axis Spectrometres. 12, 46–48, 54, 64, 244–252

MinGW Minimalist GNU for Windows. 157

MON Monitor Counter. 29, 255

NBI Niels Bohr Institutet Kobenhavn. 46

NBM NetBeans Module. 94

NBS National Bureau of Standards. 21

NCNR NIST Center for Neutron Research. 43

NECSA South African Nuclear Energy Corporation. 2, 36, 56, 59, 263

NeXus Data Format for Neutron, X-Ray & Muon Science. 2, 36–38, 41, 42, 44, 46, 50, 51, 53, 54, 63, 66, 106, 107, 198, 213

NIAC NeXus International Advisory Committee. 50, 198

NISP Neutron Instrument Simulation Package. 47, 244

NIST National Institute of Standards and Technology. 21, 43

NOBUGS New Opportunities for Better User Group Software. 83, 213

NSIS Nullsoft Scriptable Install System. 156

ODT Open Document Text. 94

OI Open Inspire. 2, 13–15, 24, 36, 57, 74, 76–91, 94, 96, 98, 101, 103, 104, 106, 109, 111, 112, 114–118, 120–128, 131, 135–141, 143–152, 154–162, 164–170, 172–179, 181–189, 191–194, 196, 197, 199, 200, 202–204, 206, 209, 212–214, 216–220, 222, 228, 229, 231, 233, 237, 238, 240, 242–245, 249–252, 254–257, 265–278

OI-LAMP Open Inspire Library and Media Platform. 78, 85, 91, 109, 111, 116, 118, 127, 128, 131, 134–136, 150, 161, 162, 166, 167, 169, 170, 172, 274

OIL Open Inspire Library. 89, 96, 102–111, 115, 116, 134, 136–138, 151, 154, 159, 161, 166, 167, 169, 170, 172, 173, 175, 177, 199, 200, 268, 269, 271, 272, 276

OIM Open Inspire Module. 86, 89–96, 98, 99, 101–106, 109–111, 114–116, 118–121, 123–128, 130, 131, 133, 134, 136–146, 150, 151, 154, 155, 159, 161, 162, 166–173, 175–178, 182–186, 188–190, 192–197, 199–204, 208–213, 217, 218, 220, 221, 228, 230–232, 236, 237, 242–252, 254, 255, 257–259, 262, 266–274, 277, 278

omgs omega-s Rotation Axis. 28, 31, 208, 258

Open Inspire Open Infrastructure for Natural Science and Programming in Research Environment. 11, 12, 14, 15, 30, 32, 54, 57, 73, 77, 86, 87, 89, 106, 107, 114, 115, 119, 124, 125, 128, 144, 146, 147, 149–152, 159, 160, 168, 209, 211

ORB Object Request Broker. 215, 222, 223, 230

ORM Object Relational Mapping. 49

ORNL Oak Ridge National Laboratory. 48

OS Operating System. 31, 49, 80, 106, 107, 136, 137, 149, 150, 156, 157, 160, 173, 174, 181, 197, 245, 270, 282, 306, 312, 313

OSGi Open Services Gateway initiative. 38, 39, 88, 94

OSI Open Source Initiative. 65, 320

OSX Mac OS 10. 275

OU Open University Milton Keynes. 2, 56

PC Personal Computer. 218, 253

PDF Portable Document Format. 15, 154, 274

POJO Plain Old Java Object. 98

PSD Primary Slit Distance. 27, 261

PSI Paul Scherrer Institut. 35–37, 43, 46, 47, 56, 59

PSR Primary Slit Rotation. 27, 262

PST Primary Slit Translation. 27, 261, 262

PV Process Variable. 39

RCP Rich Client Platform. 38, 39, 115, 175, 191

RegEx Regular Expression. 123, 124, 133

RMI Remote Method Invocation. 145, 184, 185

RNL Riso National Laboratory. 47

RPM RPM Package Manager. 157

SCADA Supervisory Control and Data Acquisition. 58, 74, 76, 78, 85, 116, 291

SDET Single Detector. 29, 255, 260, 261

SDS Scientific Dataset. 51, 200

SE Standard Edition. 88

SEI Studiengruppe für Elektronische Instrumentierung. 216

SICS SINQ Instrument Control System. 36, 37, 54, 77, 243

SNS ORNL Spallation Neutron Source. 40

SOAP Simple Object Access Protocol. 256

SQL Structured Query Language. 49

SRS Software Requirements Specification. 55–58, 265, 266

SScanSS Strain Scanning Simulation Software. 40, 64, 209–211

SSH Secure Shell. 66, 185, 191

SWT Standard Widget Toolkit. 174

TCP Transmission Control Protocol. 185

TDD Test-driven development. 62

tths two-theta Rotation Axis. 29, 31

TUM Technical University Munich. 24, 56

UI User Interface. 12, 19, 34, 69, 78, 79, 146, 147, 149, 174, 175, 181, 182, 186–189, 191, 194–196, 202, 209, 251, 252, 266, 268, 272, 276, 277, 281, 282, 293, 296, 297

UML Unified Modeling Language. 14, 61, 179

URI Uniform Resource Indicator. 92, 95

URL Uniform Resource Locator. 13, 15, 110, 150, 157, 171

UTF-8 8 Bit Unicode Transformation Format. 117, 285

VCS Version Control System. 85, 154–156, 158, 161

VI Virtual Instrument. 41

VITESS Virtual Instrumentation Tool for Neutron Scattering at Pulsed and Continuous Sources. 46–48, 244

VLAN Virtual Local Area Network. 66, 218

VM Virtual Machine. 83

VME Versa Module Eurocard. 30, 31

WAR Web Application Archive. 94

WWW World Wide Web. 92

WYSIWYG What You See Is What You Get. 40

xE x-Translation Axis for Eulerian Cradle. 28

XML Extensible Markup Language. 53, 97, 98, 101, 103, 115–118, 133, 144, 191, 196, 198, 199, 202, 208–210, 268, 273, 274

xT x-Translation Axis. 28, 31

yE y-Translation Axis for Eulerian Cradle. 28

yT y-Translation Axis. 28, 31

zT z-Translation Axis. 28, 31